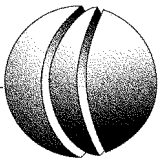
1



Components Everywhere

Water, Water, every where, Nor any Drop to Drink.
—Samuel Taylor Coleridge
The Rime of the Ancient Mariner

The belief that using commercial components will simplify the design and implementation of systems is widely held, but is, unfortunately, belief in a compelling myth. Imagine you are the architect or chief engineer for a project that is developing a large, mission-critical information system. For whatever reason, it has been decided that the only practical—though a very promising—means to this end is to build the system from commercially available components. Before the project makes substantial progress, though, it is confronted by the obstacles that follow:

- Many crucial design decisions revolve around component interfaces that are deficient in several respects, but are not subject to negotiation. Yet these same interfaces are often unexpectedly modified by their vendor, simultaneously invalidating previous design workarounds and introducing new integration difficulties.
- The components have been designed to be easy to integrate with some vendors' components but difficult to integrate with others. Unfortunately, the best components reside in different vendor camps. You discover that the question of which components to use is more complex than anticipated, with each selection decision contingent on some set of other selection decisions.
- The components that have been selected are sufficiently complex that no one on the project knows exactly how they work. When integrated, the component assembly exhibits baffling and incorrect emergent behavior. The vendors have never seen this behavior before, but they are *certain* the fault lies with another vendor's component.

Does this sound familiar? How did this state of affairs come about, and what can practicing designers and software engineers do about it? Can predictable and

repeatable software engineering methods be established in the face of these challenges? We believe the answer is yes.

1.1 The Software Component Revolution

Software components are everywhere—or perhaps we should say this about magazine articles, scholarly papers, market forecasts, and (yes) books about software components. Judging by all that has been written, and continues to be written, the software industry is quickly moving toward component-based development. Add to this the avalanche of commercial software methods and tools that support component-based development and the conclusion seems inevitable.

A closer look at the evidence, however, presents a picture that is far less clear. We do not argue against this trend. Instead, we propose that this trend, far from being unified and coherent, is a splintering of many trends, some competing, some reinforcing, and some wholly independent. Organizations interested in adopting component-based development for competitive or other reasons are immediately confronted by a nearly impenetrable and fast-growing thicket of technological, methodological, and organizational options.

To state the conclusion first, software components are real and are already profoundly altering the practice of software engineering. Despite signs of progress, the challenges posed by systems that are constructed predominantly from *commercial off-the-shelf components* have yet to be addressed adequately. Indeed, the more general challenges posed by large-scale reuse of existing components, regardless of their origin, have yet to be addressed. Further stipulating the commercial origin of components merely adds further complexity to these already unaddressed challenges.

To be sure, there are several accepted and well-defined component-based development methods, notably Cheesman and Daniels' UML Components [Cheesman+ 00], D'Souza and Wills' Catalysis [D'Souza+ 99], and Herzum and Sims' Business Component Factory [Herzum+ 00], each recently published and each a truly excellent contribution in its own right. However, these methods assume that the design task is, predominantly, one of component specification for later implementation rather than one of component selection for later assembly. Only Herzum and Sims recognize this apparent contradiction between the premise of currently published component-based development methods and the *consequence* of adopting a component-based paradigm; namely, that applications would be assembled from existing components rather than from custom-developed components. Herzum and Sims address this by describing a reuse process (the business component factory) that shifts, ever so gradually, the emphasis from custom development to reuse of existing components.

But surely this misses two key points. First, design methods that assume the freedom to define component interfaces are fundamentally different from methods that lack this freedom.[1] Defining a method for the former condition in the hopes of transitioning this same method to the latter condition is doomed to failure. Second, there are many systems being built today whose fundamental design challenge stems from an aggressive use of commercial software components. By commercial components we mean things such as Web browsers, HTTP servers, object request brokers, relational database management systems, message-oriented middleware, public key infrastructure, transaction monitors, and geographic information systems. Software engineers need design methods to deal with this class of component-based system *now*, not in some indefinite future.

Our book addresses this gap in software engineering methodology. In particular, we define concepts, techniques, and processes to address the design challenges uniquely posed by the use of commercial software components. These include, but are not limited to, the challenges that follow.

- Critical design decisions must be made in the face of incomplete and often mistaken understanding of the features and behavior of commercial software components. Knowledge gaps are inevitable and are a major source of design risk.
- Whatever knowledge is obtained about one commercial software component does not translate easily to components from different vendors, and all component knowledge tends to degrade quickly as components evolve through new releases.
- Competitive pressures in the software marketplace force vendors to innovate and differentiate component features rather than stabilize and standardize them. This results in mismatches that inhibit component integration and inject significant levels of wholly artificial design complexity.
- Use of commercial components imposes a predisposition on consumers to accept new releases despite disruptions introduced by changing component features. These disruptions take on a random quality across all phases of development as the number of components used grows.

These challenges all derive from the same root cause: a loss of design control to market forces. Perhaps the reason why no methods have yet been developed to address these challenges is that we have been trained to think of market forces as beyond the scope of software engineering methods. But it is a poor engineering method that fails to accommodate practical reality. These market

[1] Design *with* interface freedom reduces to a form of continuous optimization tradeoff wherein each component interface represents a variable that can be arbitrarily modified to produce more or less optimal results. Design *without* interface freedom reduces to a form of selection tradeoff wherein each component interface represents one choice among a discrete set of choices. These two modes require fundamentally different approaches to design.

forces play as much a part in software engineering as friction plays in mechanical engineering.

Before delving into details we must first set some context. We describe the class of design problem addressed by this book, especially with respect to different conceptions of *software component* and *component-based development*. Then, we outline our assumptions about the software engineering methods and processes that we expect to be in place already. Lastly, we deal with terminology.

1.2 Component Space

Although the term "component-based development" does not refer to the development of just one type of system, or to the use of just one type of component, most conceptions of component-based development share a few fundamental concepts. Foremost of these is that components are software implementations that have interfaces and that are units of independent substitution. But beyond this, there are numerous variations, both large and small. We have found three dimensions of variation to be particularly apt for describing the big picture.

The first dimension concerns the source of software components. As we noted earlier, published component-based software methods share a premise: that the principal task of the design effort is to produce component specifications. This premise is clearly invalid in situations in which most or all of the components already exist. The distinction we make, then, is between development efforts that specify their own custom components and efforts that are constrained to use only preexisting components. Today, the most important source of preexisting components is the software component marketplace. It is the marketplace that leads to economies of scale and that truly differentiates component-based development from other software engineering paradigms.

The second dimension concerns the environments into which components are deployed and in which they execute. Again, there are two major distinctions. Some components are deployed directly onto a native operating system. This class of component need only comply with the interfaces and conventions imposed by that operating system. Other components are deployed into a higher-level environment that is variably referred to as a component framework [Bachmann+ 00] [Szyperski 98], business component virtual machine [Herzum+ 00], or component standard [Cheesman+ 00]. Whatever it is called, the distinction between framework-based and operating system-based components is an important one, since a component framework constrains component developers and simplifies component integration.

The third dimension concerns the use of components to implement *application* versus *infrastructure*. Obviously this distinction is subjective: the best definition of infrastructure that we know of is "whatever it is that I need to do my job."[2]

[2] We are indebted to Len Bass for this definition.

Consequently, one engineer's infrastructure may well be another engineer's application. Nevertheless, the idea of infrastructure does have substantive meaning for one very large class of system—the enterprise information system. Enterprise information systems tend to be, among other things, large, heterogeneous, distributed, multi-user, persistent, transactional, and secure. By infrastructure, we mean the software that implements this functionality. In contrast, enterprise information systems use the infrastructure to implement and deliver business services to users.

Some people refer to infrastructure dismissively as "plumbing" as a way of suggesting that this functionality *should* be assumed to exist within a component framework, that the problems of infrastructure are, by and large, solved, and that the *real* software engineering problems reside elsewhere. We agree that one of the benefits of component frameworks is that they bundle infrastructure services [Seacord+ 99]. In principle, we agree with Herzum and Sims (and many others) that component technology will evolve in the direction of more complete and robust frameworks. However, we do not expect one standard framework to emerge, but rather a variety of component frameworks, each tailored to its own requirements. Such frameworks will themselves need to be integrated, and new releases of frameworks must be installed and integrated. There is no escaping infrastructure, and building and sustaining an enterprise infrastructure is an excruciatingly complex undertaking. Software engineering methods are needed here, too.

Figure 1-1 depicts these as three orthogonal dimensions in a Cartesian coordinate system. We, of course, understand that the world of components is far more complex than can be accommodated by these three dimensions. But the resulting picture of "component method space" adequately situates the subject matter of this book with respect to different classes of component-based system and their related development methods.

As mentioned earlier, Cheesman and Daniels' UML Components targets the design of applications built using frameworks (upper right rear of figure). Although their method includes a step to search for existing components, doing so *after* defining component interfaces practically guarantees that no components will be found. Herzum and Sims are even more aggressive in postulating component frameworks than Cheesman and Daniels. They assume frameworks exist at each conceptual layer (user, workstation, enterprise, and resource) in their canonical system design. Both of these methods focus on application rather than infrastructure. That is, they assume the existence of an infrastructure rather than describe a method to construct one. Likewise, both methods assume the design task is predominantly one of specifying components rather than assembling systems from existing components. D'Souza and Wills' Catalysis, however, differs from UML Components and the Component Factory in neither assuming nor rejecting component frameworks. In this respect, Catalysis has broader conceptual applicability. On the other hand, Catalysis does not address the technical infrastructural aspects of enterprise systems that are made explicit, albeit only assumed, in the other methods just discussed.

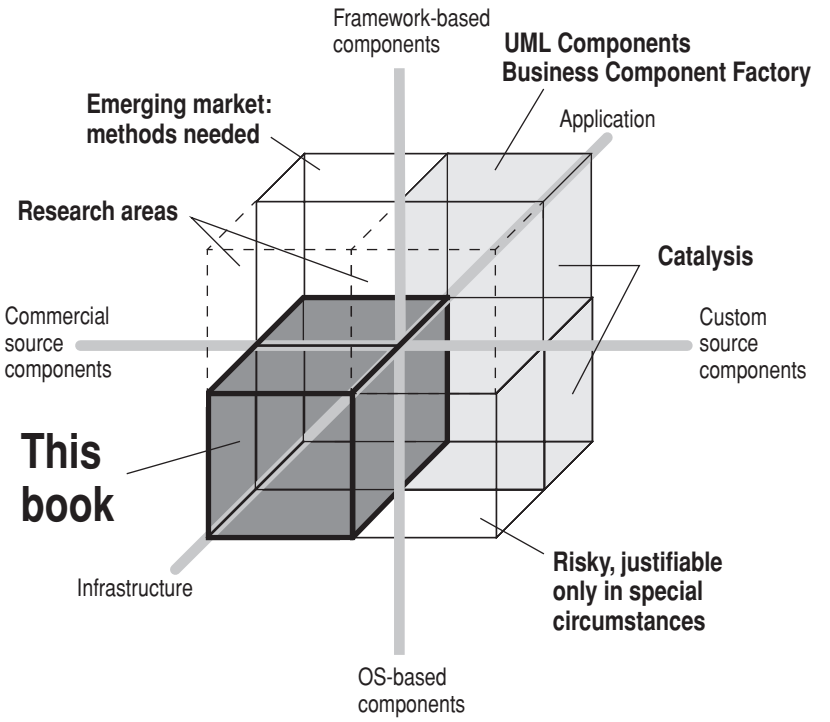


Figure 1-1 The structure of component method space.

The region of framework-based commercial components used for applications (upper left rear of figure) represents an important emerging market. It includes, for example, markets in commercial off-the-shelf Enterprise JavaBeans. Such components are already available in the commercial marketplace as stand-alone components and as product lines of components. While the component framework used in such applications distills away much of the integration complexity that is tackled by this book, the market forces that drive the component market ensures that a considerable residue of mismatch among components will remain. As a result, we expect that engineering methods that address this class of application will need to combine elements described in this book with what is described by Cheesman and Daniels. In any event, work is needed in this area, and soon.

We have dash-outlined the two uncharted regions of space corresponding to framework-based components for infrastructure. There are research projects (sometimes referred to as "programmable middleware") that are under way in this area of space. While these efforts might produce industrial-grade results, these regions are essentially unexplored. Note one area of space that should be avoided at all costs: enterprise infrastructures built from custom software. There

may be some justification for this for highly specialized domains, but otherwise there is little justification for avoiding commercial technologies.

The remaining regions of component-method space—applications and infrastructure constructed predominantly from commercial-off-the-shelf (COTS) components—are the subject of this book.

1.3 Process, Method, & Notation Assumptions

While commercial software components pose unique challenges to software engineering, they do not require that we revisit each and every precept of theory and practice. To be sure, certain software engineering practices need to be adjusted, and some outdated notions of software process need to be revised substantially. Some of these adjustments and revisions will not be easy to undertake given how entrenched these precepts have become. Given this challenge, it certainly makes sense to minimize difficulties by starting from a reasonably well-established foundation. Accordingly, we make some assumptions about software development processes as context for the material discussed in this book.

Our first assumption is that all projects use some form of Boehm's spiral development [Boehm 88a] [Boehm+ 00]. Spiral development is characterized by iterative development, with risk analysis at the inception of each iteration. It is important to understand that spiral development is not just best practice, but *essential* practice for building systems from commercial components. Of course, what Boehm actually described was a *metaprocess*—characteristics that any particular software development process must possess for it to be a spiral process. Therefore, it is not sufficient for us to say only that we assume the spiral model, we need to be more specific.

We assume as a starting point the use of the Rational Unified Software Process (RUP), and we use the terminology and idioms of RUP throughout this book. We are well aware of the shortcomings of RUP, and are reminded of a story about President Abraham Lincoln. When an angry congressman insisted that any general would be better than the one Lincoln had appointed, Lincoln responded by saying that he could not appoint *any* general, he had to appoint *some* general. Although RUP is biased toward an object-oriented conception of the world (which is fine if you are doing object-oriented development, of course), and although it is strategically vague in a number of areas, it has several things going for it.

- It is defined.
- It is extensible.
- It supports spiral, iterative, and incremental development.
- It defines a variety of developer roles and their related tasks.
- It does not inhibit the use of components (though it does not address it).

For these reasons RUP provides a good place to start. Of course, this does not require that *you* use RUP. It only means that we find it useful to appropriate RUP terminology and process models as a means to illustrate ideas.

We also have chosen the Object Management Group's Unified Modeling Language (UML) as our notation. Although UML lacks certain features that would make it more effective as an architectural modeling language, we have selected it because it is widely used and should be familiar to most readers.

1.4 Terminology and Acronyms

In the remainder of this book we use the term *component* to mean *commercial software component*. *Component-based development* refers to the practices required to build systems (applications and infrastructures) from commercial software components. This is not to diminish the claims of the other authors cited as to their component orientation. It is, however, an explicit rejection of any claims to the exclusivity of that term.

You will also note that we do not construct many categorical definitions. Of course, it is important to be clear about certain key concepts, and we do (finally) define software component in Chapter 5. But it is difficult to construct airtight categorical definitions. Indeed, it is not even necessary; we can all identify and use money without knowing precisely how to define it.[3] Our preference is to use definitions sparingly, and instead illustrate how the categories are used. This is far more satisfying for the authors and, we expect, for readers as well.

Also, sadly, software engineering and especially the commercial software marketplace generates a seemingly infinite variety of acronyms, nested acronyms, and compound acronyms. Rather than redefine the same acronym in each chapter, we do so only once, and ask you to refer to the glossary for all other uses.

1.5 Summary

Industry is rushing pell-mell to adopt component-based development. But the challenges posed by a large and important class of component-based system, those comprised of commercial software components, have remained unaddressed, or certainly underaddressed. This book establishes the foundations for a methodological response to these challenges.

[3] This is an example of the so-called *Socratic fallacy*, which asserts that it is impossible to know whether something is an instance of a property unless one has a definition for that property. Incidentally, Socrates was, in fact, *not* guilty of falling prey to the fallacy that bears his name.