Book Review

Component Software: Beyond Object Oriented Programming, Second Edition
by Clemens Szyperski

Addison-Wesley, 2003
ISBN: 0-201-74572-0
Cover Price: US$54.99
608 Pages

What are software components? We all have a preconceived notion of what they are but have difficulty putting our thoughts into words. It is very challenging to define something that is so abstract. That is why Clemens Szyperski, a well-established researcher and writer on the subject of component technology, has taken 589 pages to explain it. In the very beginning of his book he states that:

> Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system.

Don't worry if you don't understand this, because the book goes on to explain in great detail what components can be and how they are used in today's industry. First, Szyperski introduces the basic theory behind components; then he covers many technological practices, including object-oriented principles and Web services; and finally, he closes by introducing even more theory and then wrapping everything up in a conclusion.

The book remains demanding, though. Szyperski warns readers early on that some of the upcoming passages are difficult reads, and I did find that some of his points went way over my head. He targets mainly computer theorists, systems architects, and integrators, software developers, and CTOs, and assumes familiarity with object-oriented principles. As the book focuses on theory rather than code, it is not a good reference for specific languages such as C#. It does, however, explain why programmers use each specific language. I definitely recommend it for anyone searching for an in-depth look at today's component technology. If you have a fear of computer-related acronyms, *Component Software* will become an invaluable tool for you.

Are Objects Components?

In his opening discussion of component theory, Szyperski posits that a component has three characteristic properties:

- It can be deployed independently.
- It is a unit of third-party composition.
- It has no (externally) observable state.

Since most programmers today use object oriented-based languages such as Java and C++, they intuitively think of objects and libraries as components. There is little dispute that libraries should be classified as components, but there is much debate over whether objects are components. Szyperski says that an object:

- Is a unit of instantiation; it has a unique identity.
- Unlike a component, may have state, and this can be externally observable.
- Encapsulates its state and behavior.

Note the conflict: Components cannot, technically speaking, have externally observable states. So, although some objects can *act* as components, it is not accurate to classify all objects as components, Szyperski explains. Also note that not all objects used in an application are from third parties; many are developed in-house. Just because an object can be instantiated many times within an application does not necessarily mean it will have any functional value within another; in other words, it is not necessarily a reusable component. Objects created by third parties are typically contained within libraries, which are built for reuse. *Those* objects are considered to be components -- but again, as only some objects act as third-party components, it is simply inaccurate to classify all objects as components.

The Component Dilemma

To be a component or to not be a component. That is the big question Szyperski poses in the theory section. Basically, he argues that if you can't foresee reuse for a piece of software, then it may be more efficient and cost-effective to create custom methods.

Typically a component includes an interface that a client uses to communicate with the component. If the component is based on a black box model, the client may have very little knowledge of the component's inner workings, which makes the interface crucial to understanding what the component does and how to communicate with it. The problem is, as Szyperski explains it, that the more specifically a component's function addresses your particular needs, the more intricate the interface becomes; this can result in less potential for component reuse and unjustifiable development costs, especially if you need only a small portion of that functionality yourself. The unofficial "standard" in the component field says that a component must be used at least 2.5 times in order to prove its

value and justify your investment. If you can't foresee this much reuse, then it may not pay to build a component interface.

On the flip side, if your component has less functionality and an easier interface, you may need more components to perform your desired task, and tracking them may become an even larger challenge. The key to creating good component architecture is to find a happy medium and create components that are both manageable and reusable.

Comprehensive Technology Coverage

The technology section, which comprises most of the book, is dedicated to specific uses of components with today's technology and principles; it even discusses components in the software market. Szyperski starts with principles such as inheritance (how to avoid it), polymorphism, and subtypes and explains how they come into play when you build components. Later, Szyperski compares his own definition of components with those of other industry experts such as Rational's Grady Booch. In his book, *Software Components with Ada: Structures, Tools and Subsystems* (1987), Booch includes this definition:

> A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction.

Written more than sixteen years ago, this reflect's Booch's visionary thinking, although the definition does not acknowledge environmental dependencies or require that a component be independently deployable.

The most useful part of this section is near the end, where Szyperski analyzes how some of the industry's leading technologies -- J2EE, .NET, CORBA, COM, SOAP, and XML -- relate not only to components and Web services, but also to each other. He explains the purposes behind Servlets, Enterprise JavaBeans, Swing, AWT, and different Java platforms such as J2EE, J2SE and J2ME. He does an excellent job of summarizing .NET, J2EE and Component Pascal and explaining the different ways that companies cope with designing component architecture. Though Szyperski now works for Microsoft, I detected no evidence of bias toward Microsoft products.

A Definition?

Don't expect to find a conclusive definition of components in this book. That is not Szyperski's purpose. Instead, he provides a great deal of helpful detail about the theories, people, and market forces driving component software. At the end of the book, I felt satisfied. Szyperski showed me why it is impossible to pin down a single, formal definition of a component, and he did what was appropriate: educate the reader and leave the issue open for further discussion.

-**Jeff Livingston**
Rational Software
IBM Software Group

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!