GREG OLSEN, COGHEAD

Ten years ago, the term *component software* meant something relatively specific and concrete. A small number of software component frameworks more or less defined the concept for most people. Today, few terms in the software industry are less precise than *component software*. There are now many different forms of software componentry for many different purposes. The technologies and methodologies of 10 years ago have evolved in fundamental ways and have been joined by an explosion of new technologies and approaches that have redefined our previously held notions of component software.

An Amazon search for the term yields more than 500 titles. Countless companies and open source projects use *component software* in describing what they do. Any summary or historical overview of the subject of component software rightfully belongs in a multivolume book. This article, then, simply looks at the highlights of how the

From COM

to Common

Component software's 10-year journey toward ubiquity

From COM to Common

concept of component software has evolved, and what we might expect in the future.

I found a somewhat dated definition of *component software* on Wikipedia that is consistent with my memory of 10 years ago:

Sometimes called *componentware*, software designed to work as a component of a larger application. A good analogy is the way personal computers are built up from a collection of standard components: memory chips, CPUs, buses, keyboards, mice, disk drives, monitors, etc. Because all of the interfaces between components are standardized, it is possible to mix components from different manufacturers in a single system.

Similarly, the goal of component software is to standardize the interfaces between software components so that they too can work together seamlessly. Two standards—OLE and OpenDoc—are designed to help programmers develop components that can work together. Many analysts believe that component software is the natural extension of object-oriented programming and that it will become the standard programming paradigm for years to come.¹

The definition is quite imprecise, but it captures the core motivations behind the concept. A more recent definition taken from Wikipedia states:

A software component is a loosely defined term for a software technology for encapsulating software functionality. Clemens Szyperski and David Messerschmitt give the following five criteria for what a software component shall be to fulfill the definition:

- Multiple-use
- Non-context-specific
- Composable with other components
- Encapsulated (i.e., non-investigable through its interfaces)
- A unit of independent deployment and versioning

A simpler definition can be: A component is an object written to a specification. It does not matter what the specification is: COM, JavaBeans, etc., as long

as the object adheres to the specification. It is only by adhering to the specification that the object becomes a component and gains features like reusability and so forth.²

These definitions in a literal sense worked equally well in 1996 as they do today. Similarly, the core motivations behind component software have remained fairly consistent:

- **Reuse.** The reduction of effort through the use of components as “standard” parts.
- **Integration.** Facilitating the interaction of independently constructed software modules on a single computer or across multiple computers over a network.

What has changed drastically over the past 10 years is our understanding of how component software concepts map to software engineering practice.

MY 10-YEAR JOURNEY WITH COMPONENT SOFTWARE

In 1996, I led a team charged with architecting a new distributed application system. At that time a component software approach generally meant using COM or CORBA technologies. As adherents of a new religion called Java, however, we decided to make do without many of the component services COM and CORBA provided, and we built out our component framework on top of a relational database and the JVM (using Java/RMI for object invocation). Our framework included basic component management and registration features and a data abstraction layer based on SGML (and subsequently XML). We also developed a container component that could provide resource and life-cycle management for a subclass of other components that we called *adapters*. In addition, we integrated third-party scripting components: one COM-based and one Java-based.

Eventually, Java beans, XML libraries, and application servers came on the scene and provided many of the features we had previously implemented ourselves. By 2000, we had evolved our product to run inside of an EJB (Enterprise JavaBeans) container—in part, to take advantage of services provided by the application server's component framework and, in part, to meet customer and business partner expectations. While there were benefits, we also discovered drawbacks to the use of the framework. Our once fairly simple custom components had become more complex—simply to meet the requirements of the framework. Work related to framework overhead seemed to become an increasingly larger portion of our total effort. Our tooling and testing infrastructure also grew substantially in complexity. The greatest source of

frustration was the knowledge that much of the framework overhead tax we were paying was to support needs that we didn't have.

In 2003, several of my former team members and I started working on a new product. The predominant component frameworks at that time were .NET and EJB. In contrast to 1996, however, many other variations on these options were available. What's more, our choice was not limited to a single framework, but included purpose-specific component frameworks that worked within other frameworks. Even the development tooling we chose to use (Eclipse) was itself a rich component software framework. Also significant was the large number of prebuilt, useful, and low-cost components that we could incorporate into our product.

Since 2003, the growth in new types of component systems and in the numbers of useful components has only accelerated. The starkest contrast between the product we are building today and the one we built in the late '90s is in the high percentage of the functionality attributable to components that we embedded but did not write ourselves.

Over these 10 years, my perspective on component software changed dramatically. Three trends, in particular, stand out:

- A redistribution of complexity—the movement of complexity out of custom-developed components and into the framework.
- Special-purpose component frameworks and frameworks within frameworks.
- Components moving to the masses—accessibility of component technology to ever-larger audiences.

REDISTRIBUTING COMPLEXITY

Back in 1996, most of the complexity of software systems lived within custom-created components. The component frameworks provided very basic services such as a registry for components and invocation brokering among components. The introduction of application servers to the frameworks allowed responsibility for some of the most complex parts of software development such as threading, memory, and transaction management to be moved out of the custom components and into the framework. Some of these benefits, however, were offset by new interface and other requirements placed on component design in order to use the new services provided by the framework. Overall, the rapid addition of framework-provided services outweighed the associated overhead and the added complexity they brought to component development.

In the past three years, however, component systems have made great strides in simplifying the construction of custom components. The components written for EJB or .NET containers are much simpler than they used to be. Newer, lighter-weight containers are taking the trend even further. A significant trend in the Java world is the use of POJOs (Plain Old Java Objects) as the basic component building blocks, as opposed to special and more complicated structures such as EJBs. A whole new class of lightweight containers (e.g., Spring, Pico, Nano) use the Dependency Injection pattern (also referred to as Inversion of Control) to shift complex responsibilities such as

MORE

Related articles in ACM's Digital Library:

Baniassad, E. L. A., Murphy, G. C., Schwanninger, C., Kircher, M. 2002. Managing crosscutting concerns during software evolution tasks: an inquisitive study. In *Proceedings of the 1st International Conference on Aspect-oriented Software Development* (April).

Bastide, R., Palanque, P., Sy, O., Navarre, D. 2000. Formal specification of CORBA services: experience and lessons learned. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)* (October).

Gill, N. S. 2006. Importance of software component characterization for better software reusability. *ACM SIGSOFT Software Engineering Notes* 31(1).

Hauck, F. J., Kapitza, R., Reiser, H. P., Schmied, A. I. 2005. A flexible and extensible object middleware: CORBA and beyond. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware (SEM)* (September).

These articles will be available online at www.acmqueue.com for an eight-week period beginning June 1. Want full access? Join ACM today at www.acm.org and sign up for the Digital Library.

From COM to Common

object creation and linking from the components to the container.

AOP (aspect-oriented programming) is another recent approach that focuses on transferring common concerns out of individual components and into the framework. The particular focus of AOP is on concerns that cross-cut (or cut across) many components in a system. The prototypical example is logging. Rather than have every component implement logging support, a centrally implemented capability is woven, or joined, into all the components. With an AOP approach, other concerns such as security or transaction support could potentially be added to a set of components via the framework.

These trends are not limited to .NET or Java-related frameworks. ROR (Ruby on Rails) adherents are religious about making Web application development easy. The Rails approach is about maximally pushing responsibility for the application into the framework and letting the developer write as little code as possible to achieve the desired result.

In 1996, the core component of the leading component framework (CORBA) was called a *broker*. This terminology reflected a conceptual view of communities of powerful and autonomous components interacting as peers, with the framework acting only as a facilitator. In 2006, *container* is the descriptive noun of choice, and the conceptual view is one where the framework provides a rich, nurturing environment to a community of minimalist components that perform focused tasks and that live blissfully ignorant of the many responsibilities that the framework is managing for them.

SPECIAL-PURPOSE FRAMEWORKS AND FRAMEWORKS WITHIN FRAMEWORKS

One reason that newer component systems such as ROR can achieve greater simplicity than some of their predecessors is that they are more narrowly focused in what they do. COM and CORBA aimed for generality. They were attempts at an Esperanto for interacting components of all potential types. In 1996, there seemed to be an implicit assumption that any component software framework had to address any and all software development challenges that might arise. In addition, early

frameworks were driven by the feature requirements of very large, complex systems expected to operate for very long periods of time.

Software development today is driven by many smaller projects that serve narrower needs. Special-purpose component frameworks that offer a simpler programming model have therefore flourished. ROR is an example of a framework that is specific to a particular type of application—in this case, fairly simple, relational database-backed Web applications. This application pattern is extremely common and is served by many other component-based approaches such as PHP/MySQL, ASP.NET, WebWorks, and JSF.

In addition to component frameworks that are specific to a particular application type, a variety of frameworks are now available that are specific to some particular function. Many component frameworks, for example, focus primarily on user interaction. Much of the current Web 2.0 hype has to do with rich client component frameworks such as the many AJAX/Flash-based frameworks. These frameworks provide building blocks specific to a user interface and to handling communication with the server side of a distributed application. The server side of the implementation may use a completely independent component framework for its implementation.

On the server side of Web applications, frameworks within frameworks are commonplace. An Apache Web server may load any number of components such as a PHP or Perl module. That module may then, in turn, load PHP- or Perl-specific modules to accomplish a specific task. New components can be added at both levels.

Eclipse is an interesting component framework example in that it provides an application- and function-specific framework and is at the same time a functioning application. The Eclipse framework is specialized for the client side of IDE-oriented applications. At one time, nearly every development-oriented application had its own IDE-esque client. Today, many applications simply add plugin components to Eclipse and use it to serve the client needs of the application. Eclipse can be used as dedicated client to a single application or can be the central client a developer uses to access many applications.

The explosion of component frameworks and complex topologies of interacting components in different frameworks has been facilitated by advances in the way that we glue components together. In the past, the method of binding components was specific to the framework in which the component lived, as was the manner in which data was passed or shared among components. This situation made it very desirable to have all components

live within the same framework. Interface technologies and practices have advanced considerably over the past 10 years, supporting both tightly and loosely coupled approaches across heterogeneous component frameworks.

XML data representations and Web service APIs represent the best examples of interface technologies that support loose coupling. Web service APIs provide a framework-agnostic way for components to interact. For example, they provide a convenient way for a browser-based client developed in one component framework to communicate with the server component of the application development in a different component framework. In 1996, available glue technology such as ASN.1 and IDLs (interface definition languages) made cross-framework component integration extremely difficult.

SOA (service-oriented architecture), one of the most prominent approaches of the past five years, is an example of component software practice. A service in any SOA implementation meets the component definition criteria (an arbitrary component may not qualify as a service, however). In a typical SOA implementation, the components are all implemented as services, and the framework consists of Web service APIs and a set of framework services.

Internet-based services are an increasingly important type of component used in a wide variety of software applications. In the past, if you wanted to add shopping cart and payment processing capability to your Web application, you would install the appropriate components into your Web application. Today, the option exists to have the Web application invoke Web service APIs on service provided by Amazon, and to use the Amazon service as if it were a component installed locally.

The difference between installed or service-based components can be transparent. For example, you can extend the capability of your Web browser by downloading a component to run on your local device, by linking to a remote service, or through some combination of the two—without knowing which option was actually used.

COMPONENTS TO THE MASSES

In 1996, component software was something that was directly relevant to advanced software development professionals. To use component software, you had to have a solid foundation in computer science concepts and you had to invest significant effort in understanding the specifics of whatever component framework (e.g., COM or CORBA) you were using. Today component software is something that nearly every user of software technology is exposed to and involved in.

With increased simplicity, newer component frameworks have reduced the software development skills required for their use. In many frameworks, very minimal technical skills are needed to add, configure, and use components, while greater skill is required to create new components. To add and configure components to an Apache/PHP-based Web application, for example, one typically modifies a configuration file and then invokes component functionality via scripting. Many different component-based application systems (e.g., Drupal Content Management System, SugarCRM system, etc.) provide similar capabilities.

Nearly everyone who uses a Web browser has installed plugins, add-ons, and/or extensions (and knows what these things are). Firefox lists more than 1,200 extensions and plugins and a similarly large number exists for Internet Explorer. The situation is similar for Outlook

Internet-based services are an increasingly important type of component used in a wide variety of software applications.

and other office productivity applications. Clearly these extensions are examples of component software.

There are examples of component software use that span the spectrum from advanced developer to naïve user of applications. One interesting new area of component-oriented software application development is the SaaS (software as a service) application using a service-based infrastructure. Service composition in the form of *mashup* is a type of component software development. There are now numerous service APIs from providers such as Google, Yahoo, Amazon, and eBay that are used in composition of other service-delivered applications. Today, with services such as Salesforce.com's AppExchange, Intuit's QuickBase, Coghead, or JotSpot, it is possible to create and deploy a software application without any installation of software or without highly advanced pro-

From COM to Common

gramming skills. Each of these offerings supports a form of component or service reuse.

THE COMPONENT CHALLENGE

Without a doubt, our perception of component software today is very different from what it was 10 years ago. In many ways, componentry in software now resembles to a much greater degree the componentry in mechanical and electronic systems upon which the initial vision was based. Large numbers of components exist that can be used within large numbers of component systems that can be assembled into working applications—in ways that are analogous to electronic components put on boards put into devices or mechanical parts into subassemblies and into systems. Like electronic or mechanical systems, component software systems are relevant to people of diverse technological skill levels. Today many software capabilities are delivered solely through component assembly and configuration—by people with only limited technical skills.

The component software world, however, has in many respects already exceeded its electronic or mechanical brethren. Less inhibited by the constraints of physics, component software systems can support topologies, types of components, and uses of componentry that have no electronic or mechanical analogy. Even simple software systems are being built using multiple levels of component hierarchy (i.e., components within systems within systems...). Interface abstractions such as Web services are facilitating component interactions that cross system and hierarchical boundaries. There are software systems composed of components, for example, that can look and act as a single component (even with respect to the components inside of it). Try to imagine recursion in a mechanical system. Similarly, physics significantly limits the use of centralized, shared services in mechanical or electrical systems. In software systems, opportunities abound and bring with them possibilities for order-of-magnitude improvements in efficiency.

The rise of component software, however, is challenging the practice of software engineering in unforeseen and fundamental ways. The multitude of component frameworks and the high rate of arrival of new frame-

works are making skill shortages and rapid retraining more critical than ever.

The types of skills required have also changed. Software engineering is increasingly focused on system composition challenges, as opposed to writing particular blocks of code that implement algorithms, data structures, or other functions. The capabilities and quality of software applications depend on the engineering teams' abilities to do things other than write good code (e.g., subsystem selection and provider qualification, composition of components and subsystems, and testing of heterogeneous systems). The ability to incorporate more capabilities from commercial and open source and from packaged and SaaS providers brings power and challenges. How is overall system behavior (e.g., performance, reliability, and security) assured when the system is composed of independently managed services? What are the ramifications of using components bound by various open source or commercial license models? Which standards matter, and which don't? Which providers are likely to succeed? To fail? These questions are not new, but are more critical in a composition-oriented world. Most software engineering teams today are not well equipped to answer these types of questions.

Our challenge, then, is not only to improve component software technology but also to develop the methodologies, training, curricula, and experience base needed to deploy the technology successfully. Q

REFERENCES

1. http://www.webopedia.com/TERM/c/component_software.html.
2. http://en.wikipedia.org/wiki/Component_Software.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

GREG OLSEN is the chief technology officer and founder of Coghead, a new SaaS (software as a service) provider. Previously, he cofounded Extricity, a business-to-business integration platform provider that was acquired by Peregrine Systems. At Peregrine, he directed new platform efforts as vice president of development. Before Extricity, he managed projects at EIT (Enterprise Integration Technologies), one of the first Internet technology companies. His doctoral research focused on computational support for distributed engineering teams. He holds Ph.D. and M.S. degrees from Stanford University, and M.S. and B.S. degrees from the University of California, Santa Barbara.

© 2006 ACM 1542-7730/06/0600 \$5.00