THE EVOLUTION BEYOND OBJECTS: SOFTWARE COMPONENTS

Lynessa Coutto, Gonzaga University, Spokane WA, lcoutto@gonzaga.edu
Jason C.H. Chen, Gonzaga University, Spokane WA, chen@gonzaga.edu

ABSTRACT

Software components have emerged as a key technology in the development of complex software systems. Components have shifted the focus from the development of new software to the integration of existing components to provide new functions. This paper describes component-based systems in terms of components, component models and component frameworks. It explores the advantages and the issues related to component-based systems.

Keywords: Components, Objects, Component-based systems, Model, Framework

INTRODUCTION

Component-based software development represents an important stage in the field of software engineering. As systems become larger, interdependent, and more complex, system design has begun focusing on the integration of components rather than on the development of new software. While the concept of software components has been well known, the practical aspects and challenges have more fully evolved over time (3).

This paper describes component-based systems in terms of components, component models and component frameworks. It explores the promises of component-based systems – productivity, quality, and other benefits. It finally discusses the issues related to component-based systems that concern integrated system customers.

COMPONENT-BASED SOFTWARE DEVELOPMENT (CBSD)

Software Developers have long held the belief that complex systems can be built from smaller components, bound together by software that creates unique behavior and forms of the system (3). CBSD focuses on building large software systems by integrating existing software components. By enhancing the flexibility and maintainability of systems, this approach can be used to reduce software development costs, assemble systems rapidly, and reduce the ever-increasing maintenance burden associated with large systems (6,7).

At the core of the CBSD approach is the assumption that certain parts of large software systems reappear with sufficient regularity. These common parts should be written once and be assembled through reuse rather than rewritten over and over again. This approach embodies the "buy, don't build philosophy", put forth by Brooks (1987). These systems encompass both commercial-off-the-shelf (COTS) products and components acquired through other means, such as non-developmental items (NDIs).

According to the Carnegie Mellon Software Engineering Institute (6), developing component-based systems has become feasible due to the following factors:

- The increase in the quality and variety of COTS products.
- Economic pressures to reduce system development time and maintenance costs.
- The emergence of component integration technology.
- The increasing amount of existing software in organizations that can be reused in new systems.

CBSD has shifted the development emphasis from programming software to composing software systems. CBSD has three basic requirements, according to Hopkins (3):
a) Components: There must be a ready supply of well-built, applicable components that can be licensed and easily used.
b) Component Models: There must also be a component model to support the assembly and interaction of components. It acts as a standard "backplane" in which the components can exist and communicate.
c) Component Frameworks: This involves the framework and architecture that supports component-based development.

COMPONENTS

Meyer (5) defines a component as a "natural extension" of the object model. Object-technology has been found to bridge the communication gap between system developers and business people. Business objects also tend to reduce complexity because programmers do not need to know how it works internally.

Object-technology, however, has certain disadvantages, like a steep learning curve. Objects become difficult to handle when they are combined and recombined in large-scale commercial applications. What is required are ensembles of business objects that provide major chunks of application functionality, and these ensembles should have the ability to be snapped together to create complete applications (7). The next step in the evolution beyond objects embodies this approach. It involves the development of software components.

A component is a physical manifestation of an object that has a well-defined interface and a set of implementations for the interface. However, a component may or may not have been created using O-O tools and languages (3).

Features of a Component

Based on the definitions (3,9) we can list the features of a component:
Self-contained Packages*:* Software components are self-contained packages of functionality that have clearly defined, open interfaces that offer 'plug-and-play' high-level application services.
Independence and Flexibility: Independence does not necessarily mean that a component has no dependencies on other components. In the case of components, independence means that the dependencies are generic enough to be satisfied by several different providers.
Scale and Size: Though there are no real limits, a single component provides insufficient functionality for forming a complete system. The norm involves integrating several components into a larger, complete application.

Explicit Interface: Although the inner workings of a component can be treated as a black - box, its interface must be explicitly defined. This includes a complete listing of the services provided and how to access them, generic dependencies and error conditions that might occur (9).
Granularity: As inter-component communication is fairly expensive in terms of time and platform resources, components are encouraged to be larger rather than smaller. However, larger systems tend to have complex interfaces and represent more opportunity to be affected by change (5). Thus, the larger the component, the less flexible the structure of the system. A balance has to be struck based on the likelihood of change and the complexity of the component.
Build-time Dynamism: As individual units of change, components can be swapped, replaced, or upgraded during the build of a complete application (9).
Runtime Dynamism: More complex component architectures allow component services to be looked up and hooked up while an application runs (9).
Specificity: The more closely the component matches the design, the less the required modification. However, the number of components within the system will increase, as components become more specific.
Distribution, Modularity, and Independence of platform and language: Any component can exhibit varying degrees of distribution, modularity, and independence of platform and language.

COMPONENT MODELS

A component model specifies the standards and conventions imposed on those who develop components. They specify a means for components to publish their interface, send messages and pass data. For a component-based system to work, it is necessary to have a component model on which to base the development and communication of the components (3).

Current Component Models

Some of the current component models that have gained commercial support are:
DCOM (Distributed Component Object Model) model from Microsoft is largely confined to the Windows platform.
CORBA (Common Object Request Broker Architecture) model defined by the Object Management Group is a language and platform independent specification, but has numerous language bindings and implementations on virtually all of the common platforms (3).
EJB (Enterprise Java Beans) model from Sun Microsystems provides an environment that is suited for the execution of Java components. Deciding which component model to use is a strategic decision, although these models may share certain common features.

COMPONENT FRAMEWORKS

A component framework is an implementation of services that support or enforce a component model. One common definition of a framework is “a reusable design of all or part of a system represented by a set of abstract classes and the way their instances interact.” Another definition used is that “a framework is the skeleton of a application that can be customized by an application developer”(1). These definitions are complementary, and not conflicting. The former definition describes a framework from a design perspective, and the latter describes it from a functional viewpoint.

A component framework may be thought of as similar to a mini-operating system (7). In this analogy, components are to frameworks what processes are to operating systems. The framework manages resources shared by components, and provides the basic mechanisms that facilitate communication among components. Like operating systems, component frameworks are active and act directly upon components in order to manage a component's life cycle or other resources, like starting, suspending, resuming or terminating component execution. However, unlike general-purpose operating systems, component frameworks support only a limited range of component types and interactions among these types.

Current Component Frameworks

Five major frameworks are used in the industry (7,8):
COM: COM framework supports both interoperatability and reusability of distributed objects. COM defines an application programming interface (API) to allow creation of components for use in integrating custom applications or to allow diverse components to interact.
DCOM: DCOM is an extension to COM that allows network-based component interaction. While COM processes can run on the same machine but in different address spaces, the DCOM extension allows processes to be spread across a network.
Enterprise Component Frameworks: The two component frameworks that are industry standards for building enterprise applications are Enterprise Java Beans and COM+; their architectures are based on a common architectural pattern called Enterprise Component Frameworks. The EJB specification defines a framework of servers and containers to support the EJB component model, with servers responsible for providing persistence, transaction and security services, while containers are responsible for managing component life cycle.
CORBA: CORBA is a specification of a standard architecture for object request brokers (ORBs). A standard architecture allows vendors to develop ORB products that support application portability and interoperatability across different programming languages, hardware platforms, operating systems, and ORB implementations.
Extensible Markup Language (XML): If components are suited to different platforms or environments, they can be integrated through XML and transported over a variety of transport mechanisms.

THE ADVANTAGES OF COMPONENT-BASED SYSTEMS

The strengths and opportunities associated with CBSD stem from the potential for reduced costs, and increased functionality and quality that multiple suppliers can bring. Some of the advantages of CBS are as follows:
Reusability: Similar to objects, components reduce development time because existing components can be reused to create more complex systems.
Ease of use: Most application programs are available as components, and most components can be easily integrated into a new application with little conventional programming.
No learning curve*:* As new components require little or no programming expertise, the learning curve is virtually non-existent (4).
Large range of available applications: There are components for 3D modeling, viewing and editing spreadsheets, full-featured word processing, creating and editing graphics, creating and

playing sound files, communicating with other computers through a modem or the Internet, geographical mapping, and even reading bar codes.

Reduced time-to-market: The availability of components drastically reduces the time taken to design, develop and field systems. Design time is reduced because key architectural decisions have been made and are embodied in the model and framework.

Easier maintenance: By creating a system that is highly componentized, the system is easier to maintain. In a well-designed system, the changes will be localized, and the changes can be made to the system with little or no effect on the remaining components (3).

Independent extensions: Components are units of extension, and a component model prescribes exactly how extensions are made. In some cases the framework itself may constitute the running application into which extensions are deployed. The component model and framework ensure that extensions do not have unexpected interactions, thus extensions may be independently developed and deployed.

Improved predictability: Component models express design rules that are uniformly enforced over all components deployed in a component-based system. This uniformity means that various global properties can be designed into the component model so that properties such as scalability, security, etc, can be predicted for the system as a whole.

Distributed Computing: Distributed systems are systems that rely on the aggregate behavior of loosely coupled subsystems. Through distributed computing, components can now communicate with each other even though they reside in different processes or on different systems.

Cost advantages: It is widely assumed that CBSD approach will be significantly less costly than the traditional approach of building a system from scratch, especially when using COTS products. This has been found to be extremely true when using components as databases and operating systems.

E-commerce advantages: Component-based systems provide e-commerce companies with the speed and agility they need to compete. As application components rely on a distributed computing structure, they free solution developers from dealing with the complexity of the technology involved. The world of e-commerce is dynamic and requires an ability to change rapidly. Such change has become an essential design goal and requires business and technology architecture whose components can be added, modified, replaced, and reconfigured. Component-based frameworks meet this need, by providing e-commerce with software solutions that are agile, capable of rapid bundling, unbundling, and rebundling (2).

THE ISSUES RELATED TO CBSD

There are many issues such as software product issues and process issues related to CBSD. We have described two major issues: business and technical issues, which affect the integrated system customer.

Business issues facing Integrated System Customers

Several business issues that are facing integrated systems customers are (6,7,10):

Changes in development process and philosophy: *An* organization's development process and philosophy may need to change when implementing component-based systems. System integration can no longer be at the end of the implementation phase, but must be planned early and be continually managed throughout the development process.

Responsibility chain: The nature of software makes it difficult to separate out the source of a particular fault even when an in-house team produces the elements. For CBS it will be important to establish sound methods of assigning and enforcing responsibility for parts and for the whole system.
Additional Costs: Integrating COTS products may add additional system development and maintenance costs for negotiating, managing, and tracking licenses to ensure uninterrupted operation of the system.
Payment: A number of issues relating to payment are likely to arise and it is possible that billing will become a major overhead for the component-based software industry. If, for example, customers pay for components on a per use basis, payment models will have to incorporate payment to component providers as well as to integrators. A much broader range of payment models than are used at present may be necessary in the future to accommodate both the complex webs of owners and agents as well as different purchasing and licensing models.
Future proofing: The provision of long-term support is likely to be an important factor to be considered when purchasing components. Practices such as multiple sources, or storing source code at an independent, secure repository may provide reassurance to organizations.
***Planning issues**:* Many of the problems encountered when integrating COTS components cannot be determined before integration begins. Thus, estimating development schedules and resource requirements is extremely difficult.
Component Redundancy: Vendors now commonly sell software products with built-in features. Though customers may pay more for such extra features, integrated products may become too large and complex to control.
Distributed Execution: Such systems could reduce the problems associated with system upgrades and version management. However, they can bring other difficulties in terms of system performance, dependency on the underlying communications infrastructure, and the security of commercial information.
Security and certification: Integrators will play a major role in virus checking, trust and risk assessment, and assessing the benefits and overhead of using certified components.

Technical issues facing Integrated System Customers

Conflicting requirements: A preexisting component has been written to a preexisting set of requirements. If the requirements are general, the system requirements can be made to conform to the preexisting requirements. However, if the requirements conflict with those of the new system, the designer may have to avoid using the component.
Architecture: The selection of standards and components needs to have a sound architectural foundation, as this becomes the foundation for the evolution of the system. This is especially important when migrating from a legacy system to a component-based system (7).
Standards: If an organization chooses to use a component-based system and also wants to make the system open, then interface standards need to come into play as criteria for component qualification. The degree to which a software component meets certain standards can greatly influence the degree of interoperatability and portability of the system.
Reuse of existing components: Though existing software can be reused, some amount of reengineering may be necessary on components before they can be adapted to new systems.
***External dependencies / vendor-driven upgrade problem**:* When integrating COTS components, an organization has to deal with a certain amount of loss of autonomy and the idea of

dependency on vendors. COTS component producers frequently upgrade their components based on error reports, perceived market needs, competition, and product aesthetics. An organization has to manage its requirements while simultaneously keeping abreast with the improvements in COTS products.

System evolution /technology insertion*:* Replacing one component often has rippling effects throughout the system, especially when many of the components in the system are black-box components. Often new versions of a component may require enhanced versions of other components, or in some cases may be incompatible with existing components.

Platforms: A given component may often be usable on only one platform, limiting its use to the domain of that platform, or requiring multiple implementations for each target platform.

CONCLUSION

Component-level programming offers enormous potential for growth. In spite of the issues that are being tackled, and those that yet remain to be addressed, software components are already receiving praise for their success in enhancing quality, productivity and functionality. As the component marketplace expands, customers will experience "increasing returns", as components will become more valuable as more become available.

Looking into the future, there are still important areas for development – like splitting standard applications into components, extending object-oriented programming to better support the component-level view of objects, standardizing component communication, and methods for sharing components among incompatible operating systems and hardware. The component revolution has already taken place, and many believe that it will be the wave of the future.

REFERENCES

1.Fayad, M., et al, "Building Application Frameworks", Wiley, NY, 1999.
2.Fingar, P., "Component-based frameworks for E-Commerce", Communications of the ACM: October 2000,Volume 43, November 10, NY.
3.Hopkins, J., "Component Primer"; Communications of the ACM: October 2000,Volume 43, November 10, NY.
4.Maurer, P. M., "Components: What if they gave a revolution and nobody came?", Computer, June 2000, Volume 33, Number 6.
5.Meyer, B., "The Significance of Components". Beyond Objects, SD Online, November 1999.
6.Software Engineering Institute, Carnegie Mellon University, "Component-Based Software Development/ COTS Integration", (2000), <URL:www.sei.cmu.edu/str/descriptions/cbsd.html>
7. Software Engineering Institute, Carnegie Mellon University, "Volume II: Technical Concepts of Component – Based Software Engineering", (2000), <URL:www.sei.cmu.edu/publications/documents/00.reports/00tr008/00tr008chap02.html>
8. Software Engineering Institute, Carnegie Mellon University, "Component Object Model (COM), DCOM, and Related Capabilities", (2000), <URL: www.sei.cmu.edu/str/descriptions/com_body.html >
9.Thomason, S., "What is a Component?"; Computer, November 2000, Volume 33, Number 11.
10.Vidger, M.R.; Gentleman, W.M.; & Dean, J. *COTS Software Integration: State-of-the-Art*, (1996), <URL: http://www.sel.iit.nrc.ca/abstracts/NRC39198.abs>