Component-Based IS Architecture

Chapter in
Brown, Carol V. & Topi, Heikki (Eds.). 2003.
IS Management Handbook, 8th
Edition. Auerbach Publications, Boca Raton, FL.

Les Waguespack, Ph.D.
Professor
Bentley College Department of CIS
175 Forest Street
Waltham MA 02452
781-891-2584
Fax: 781-891-2949
lwaguespack@bentley.edu

William T. Schiano, D.B.A.
Bentley College Professor of Electronic Commerce
Bentley College Department of CIS
175 Forest Street
Waltham MA 02452
781-891-2555
Fax: 781-891-2949
wschiano@bentley.edu

1.0 INTRODUCTION

A component is a building block of computer software systems. Modular construction has been the standard of software architecture for decades. What makes components intriguing now is the integration of technologies to support components, the use of components to compose business applications rather than the underlying system software, and component deployment and distribution over the Internet. These factors converge, leading to the prospect of shortened application development time and increased application software quality.

Problem decomposition is a fundamental technique in systems analysis. Through the process of decomposing the whole, we discover the units of information and function that must come together to define and achieve the behavior of the whole. This guiding principle permeates architecture and civil, electrical and electronic engineering. It is the same in software engineering employed in the construction of computer-based information systems.

Component technology is an outgrowth of the object oriented paradigm of system modeling, and it mirrors the technique of problem decomposition by encouraging system composition using a collection of interacting, but independently constructed, parts. In this context, *component* takes on the specialized meaning for which we use the term in this chapter.

This chapter discusses the definition, supporting technology, use, construction and economics of components. Our goal is to explain how building components and building applications using components differs from traditional software development practice. Components have the potential to shorten application development time and overall development costs. To capitalize on this potential, organizations must prepare their processes and personnel for component-based application building. For the component-based approach to be cost effective, organizations must identify requirements for componentization with a sufficiently broad internal or external market for use and reuse. The supply and demand side issues interact in a complex economic viability model, which extends beyond organizational boundaries.

2.0 DEFINING COMPONENTS

The industry has not yet settled on a universally accepted definition of component. In the broadest sense, *a component is an artifact of systems development manufactured explicitly for the purpose of being used in the construction of multiple systems by multiple development groups*. This definition encompasses most knowledge or artifacts reused in system building. That could range from documentation standards and templates to library subroutines and programming languages. How a component is used is a better way to define it than how it is built. What a component is depends on how it is used.

In this chapter, we focus on the role of components as building blocks in business applications – *business domain components*. We choose this focus to exclude building blocks of systems software, e.g. device drivers, graphics tool sets or database management packages. The stakeholders of system software components are hardware and operating system vendors whose concern is platform efficiency and interoperability. The stakeholders of business domain components are business users, managers, and systems analysts striving to satisfy requirements driven by business processes, business markets, and government regulations. We use the term *component* to mean business domain component for the rest of this chapter.

Component refers to an element of software that is clearly defined and separable from the system. It interacts with the rest of the system through an explicitly defined interface. Except for the interface, a user's knowledge of the component's internals is unnecessary for it to properly function in the rest of the system. Recent advances in software system architecture have enabled greater dispersion of components on networks and on the Internet and have therefore been catalysts for independent component development and their interchangeable combination. A component may be used in myriad system constructions.

Components marketed by out-of-house (third-party) producers are called *off-the-shelf components*. If a system developer can locate and use an off-the-shelf component, the cost of building a new, perhaps a one-time-only, solution for a particular requirement can be avoided. Off-the-shelf components arrive packaged, documented and tested; they pose a cost saving alternative to developing new software for each new system requirement.

2.1 COMPONENTS BASED ON BUSINESS REQUIREMENTS

Application systems for a particular organizational function (or cross-functional process) often share a variety of very similar computational requirements. Domain analysts recognize these similar requirements as opportunities to define components. Architects can use components during requirements analysis to frame an application domain. Components are stable architectural primitives to be combined or arranged to suit a business procedure or process; and then, when the procedure or process changes, to be rearranged quickly and easily without having to revisit the component definition. Systems analysts can use components as a lexicon for expressing individual application requirements. Designers can use components as predefined parts around which to structure a system. Programmers can use components as tangible building blocks to construct executable software.

At each step of software development a component represents an explicit understanding within the application domain: what information is germane, what actions are relevant, how action is triggered, and what consequence an action has. This constitutes a contract among the members of the extended development team – the component is a stable, reliable, known quantity. Honoring the component definitions and interfaces obviates understanding their inner workings. Developers are free to ignore details other than those pertaining to business domain function to the extent that the collection of components represents a complete lexicon of the application domain. Application development becomes a *construction set* exercise.

2.2 COMPONENT ENABLING TECHNOLOGY

Today's mainframe, desktop, portable, and handheld computers are more easily inter-connected than ever before. Because this connectivity is virtually continuous, systems architects often think of their systems as network-based or Internet based rather than computer-based. If a resource exists anywhere on the network-based system, it is readily accessible by other computers on that network. This *distributed computing* model is fertile ground for components. As an example, an application designer may need some computation (e.g. credit card validation, decryption, calendar arithmetic, format translation, etc.). If the designer can find a suitable service on the network, engaging that service rather than programming it from scratch may result in cost savings.

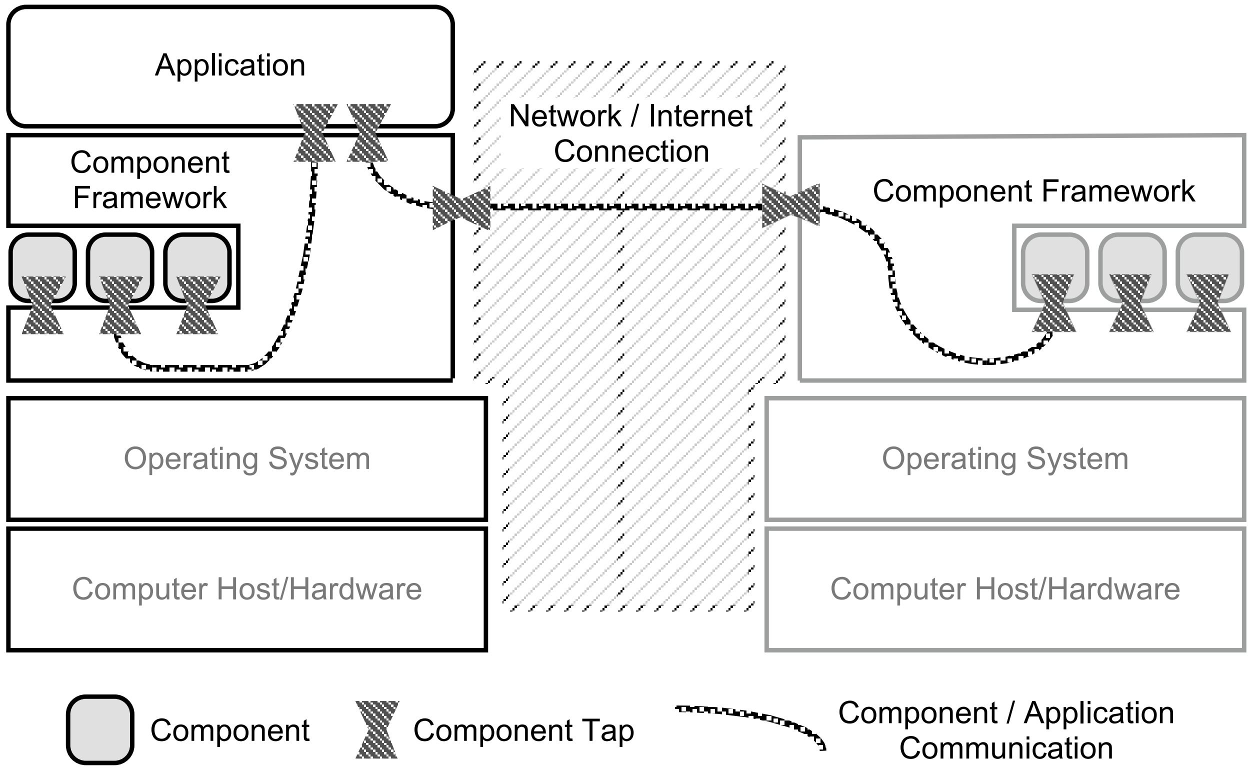

Figure 1 – Component Architecture

The technology that enables components is the *component framework*. A component framework is a combination of protocols and system software that resides on both the component server and client computer (see Figure 1 above). They handle interface connection between an application and a component. The component resides on the server computer. The application (component user) resides on the client computer. They may be the same or different computers on a network. The framework hides as many details about the component implementation, the host operating system and hardware, and the network connection as possible. A component framework provides a communication path between the application and component. Regardless of the surrounding environment, the application can ignore how the component is implemented or even where it is located and vice versa.

A component framework achieves component/application independence by providing a set of component services relating to naming, event handling, transactions, persistence and security. These services combine to provide a transaction processing environment. Component naming services permit applications and components to identify and locate one another. Once they establish communication they exchange information about their capabilities, services, and information formats. Component event services enable applications and components to get each other's attention and synchronize actions. Transaction services define collections of application activity that require all-or-nothing completion, regardless of local or remote component interaction. Component persistence services allow components to perform database management activities. Component security services, coupled with naming services, provide for client/server authentication and control access to both information objects and component functionality. All the services described herein are found to some degree in every component framework, but the developer's (or vendor's) styles and approaches differ widely.

2.3 COMPONENT FRAMEWORK PRODUCTS

This section presents a brief survey of component frameworks. We choose these frameworks as illustrative of the evolution of standards and product features over the last decade or so.
CORBA. Common Object Request Broker Architecture is a component framework standard promulgated by the Object Management Group, OMG, a consortium of major computing industry players. CORBA defines protocols, services, and an execution environment for components. OMG does not market components or a component framework. It enforces the CORBA standard and provides test suites to certify CORBA compliance. CORBA compliance assures component producers, component framework producers, and their customers that applications and components will interact reliably and consistently regardless of the vendor. OMG aggressively solicits suggestions for extensions and improvements to the CORBA standard and publishes a wide variety of supporting technical documentation and training materials. The CORBA standard has been instrumental in enabling the cross-vendor, out-of-house component market.

Microsoft ActiveX™. Microsoft has defined four standards along the way with a suite of products it markets to support components. Microsoft's industry clout has given these standards some prominence. ActiveX™ predates Microsoft's other component enabling tools. ActiveX™ uses a web browser on a client machine to download an ActiveX™ module. Once downloaded it executes directly on the client's underlying MS Windows™ machine. ActiveX™ might be described as "just-in-time downloaded programs."

Microsoft DCOM™. Distributed Component Object Model, DCOM™, is an inter-object communications protocol designed to allow components on various nodes of a Microsoft network to interact. It extends the functionality of the earlier standard, COM™, which provided similar function, but on a single host computer. Tightly coupled with Microsoft's operating system product line, MS Windows™, COM™ and DCOM™ support a thriving market in components for MS Windows™ applications including MS Office™, Visual Basic™, Visual C++™ and J++™.

Microsoft .NET™. Microsoft's most recent offering is .NET™. It extends DCOM™ adding a robust component development and configuration environment. It provides application to component connectivity, regardless of their location on a computer, a network, or the Internet. The .NET™ product suite includes program editing, compiling, and debugging in an integrated environment, Visual Studio™. Several .NET™ compliant programming languages are available to develop both applications and components.

Enterprise Java Beans™ and J2EE™. Another major player in component frameworks is Sun Microsystems. With their invention of Java™, they pioneered the "just-in-time downloaded programs" not only for web browsers on MS Windows, but for any web browser with a compliant Java virtual machine, JVM. JVMs are available for most contemporary operating systems with or without web browsers. Unlike ActiveX™, which is executable object code for MS Windows™ based computers, Java compiles into machine independent byte-code. Byte-code then executes interpretively on a JVM on the client machine. Java™, unlike ActiveX™, is both machine and operating system independent and since it is interpreted, it does not pose the security problems that downloaded executable object code presents. Java is a popular choice of developers who need "program once" and "run anywhere" software. Sun Microsystems builds upon their Java™ base with Enterprise Java Beans™ and J2EE™ that support a program editing, compiling and debugging environment for Java™ based components.

J2EE™ and .NET™ represent the state of the art in Internet-enabled, distributed component architecture technology. Each represents the evolution of component support from simply enabling components to connect and communicate, to an insulating enclosure of development, communication, database, and process management services that thoroughly enables (enforces) a model of scalable, distributed enterprise computing.

3.0 BUILDING APPLICATIONS USING COMPONENTS

Building applications using components differs from traditional application software development practice where programmers build information systems composed of tailor-made, one-of-a-kind, built-from-scratch programs. Effective use of components to build a new system requires a development process that is *component aware*. Component awareness refers to an approach in which a high value (within the organization) is placed on using components rather than building software from scratch, for 1) identifying recurring requirements suitable for component solutions and 2) streamlining processes to make component use efficient. Regardless of the specific steps, or the specific sequence, in the software process, each step requires component awareness. Either all the personnel involved in the application building process must become individually component aware, or specific personnel are assigned to oversee the use of components, or a combination of both.

In a component aware software process, requirements analysis identifies units of required functionality that are likely to be available as off-the-shelf components. Systems designers are familiar with the catalog of available off-the-shelf components relevant to their application domain. Programmers are familiar with the component framework environment that the components will plug into. Programmers are able to configure their non-component system pieces to interoperate with that environment as well. Component aware projects present a new level of challenge for configuration managers. Warranty, availability, licensing, liability, maintainability, and serviceability are all familiar issues with outsourced software or services. The configuration manager's task is, however, sorely compounded when applications and/or systems include dozens or even hundreds of components.

Each component considered, whether it is used in a project or not, exacts a cost. At a minimum, there is the cost of learning enough about the component to determine whether it is useful or not and, then, whether it is feasible to use it or not. The checklist below is representative of the information needed to consider a component for use in a particular development project. The questions show a progression of detailed knowledge needed as the project edges closer to actually using a component.

Table 1: Component Reusability Assessment Checklist

- What does it do?
    - Is it designed for our application domain or do we have to adapt it?
    - Does it support application domain functionality standards, ISO, FASB, IRS, FDA, etc.?
- Who provides it?
    - Does it come from in-house or a third party?
    - What form of warranty/guarantee does the provider offer?
    - Is the provider organizationally stable/reliable?
    - Does the provider offer configuration support services/consulting?
- What are its environmental requirements?
    - Does it require a specific component framework environment (e.g. CORBA™, DCOM™, J2EE™, .NET™, etc.)?
    - Does it require a specific configuration tool environment (e.g. Visual Studio™, Java Beans™, etc.)?
    - Does it require specific programming language expertise (e.g. C++, Java™, VB™, etc.)?
    - Is it compatible with other components we may be considering for use with it?
- How is it used?
    - Is it configurable (i.e. parameters, extension points, variable through inheritance, etc.)?
    - Does it generate source code requiring translation and/or linkage?

- ○ Does it run autonomously as a service accessed dynamically via a network/ Internet?
- ○ Are there multiple versions of the component and which is the one best for this project?

Finding candidate components and then learning the answers to these questions may represent a significant expense. To maximize the return on investment, an organization should establish its own internal catalog of the components it has examined, where they have been used, who used them, and what has been learned about them to date. The internal catalog serves as the first point of search when future component requirements arise. It is worth noting that the internal catalog will likely contain a great deal of information about components that have not yet, and may never, be used.

To control the costs of finding components and assessing their potential for any given requirement, the system's project tasks and software development life cycle need to include an explicit component information collection and management activity. These changes in organizational practice are not confined to a small group of workers, but should be widely integrated into the organization's practices. Such a broad integration relies on a strong management commitment and a disciplined project management approach. It is likely that organizations that are only casual followers of formal software engineering practices will find the task of formally maintaining a *component aware* organization very difficult.

4.0 BUILDING COMPONENTS

Just as there are organizational challenges to becoming an effective consumer of components, there are many challenges to becoming an effective producer of components. Components are intrinsically different from application software. Unlike software written as a part of a larger product, components are intended to stand alone as products themselves. They require their own life cycle management, testing, documentation and support. If they are destined for consumers outside the producer's organization, they require more sophisticated documentation, examples using them, user guides and customer support.

Choosing which component enabling technologies to use is another challenge to component producers. (See Component Enabling Technology earlier in this chapter.) By analogy, should we build a toy car part using LEGO® bricks or Tinker Toys™? The choice affects not only the pool of potential consumers and the production costs, but also has architectural and compatibility implications on the component's longevity.

An even more difficult set of choices is the selection of candidate functionality for prospective components. As we discussed in the Defining Components section earlier in this chapter, there are few limitations on what can be conceived of or defined as a component: *a component is an artifact of systems development manufactured explicitly for the purpose of being used in the* *construction of multiple systems by multiple development groups*. Which components should be built? Should we build them? Intuitively, a component should have a high probability of being used and reused – that is, high *reusability*.

Component reusability results from three characteristics briefly described below: utility, capacity and versatility.

Utility – a component's function must be relevant to a problem domain. It is common to find components that support a particular genre of system functionality (e.g. GUI services: windows, menus, dialog boxes). It is less common to find components that are application domain focused (e.g. account, client, policy, contract, agreement, etc.). Therefore, the degree of utility depends upon the domain of functionality that is a consumer's focus. The greater the utility – then the greater is the reusability. (The component does something valuable.)

Capacity – a component's function must be sophisticated enough such that using the component is clearly advantageous compared to build-from-scratch development. Searching, finding, learning, and then using a component is a labor intensive effort. It would seem that good component candidates would have some degree of complexity in their functionality along with the testing that would certify reliable and, perhaps, efficient performance. The greater the capacity – then the greater is the reusability. (What the component does is difficult to build-from-scratch.)

Versatility – a component's implementation must permit convenient integration into a target application's structure. Unless a component can be applied in a host application "as is," some configuration or adaptation is required for one, the other, or both. Adaptations become the maintenance and support responsibility of the component consumer, and these costs may outweigh the reusability benefits of the component. In the extreme, "It may be more trouble than it's worth!" The greater the versatility – then the greater is the reusability. (Although the component doesn't do exactly what is needed or how it is needed, it is easy enough to adapt for the need at hand.)

An organization's success in achieving reusability depends heavily on the producer's understanding of the consumer's problem domain. Utility depends on the problem domain almost exclusively. Capacity depends on an understanding of the architectural nature of the problem domain – which requirements are permanent and which requirements are evolving. Permanent requirements would seem to present component candidates with greater longevity potential. Versatility is strongly influenced by the choice of implementation technology and the degree of variability that a particular component must support – as in the case of evolutionary requirements.

Reusability forms a basis for a cost / benefit analysis to guide the selection of requirement candidates to implement as components. When the producer is also the consumer, an activity called *domain analysis* needs to be performed. In domain analysis, the component producer

attempts to construct a model that abstracts the requirements of prospective consumers of the component in an attempt to identify shared functionality requirements.
In the final analysis, it boils down to questions of economics: the consumer asks, "Does building systems using components increase or decrease the system life cycle costs?" And the producer asks, "Does the economic return on components produced justify the cost of building them?" These questions are explored in the following section.

5.0 MAKING COMPONENTS COST EFFECTIVE

Components may streamline systems, but using them increases the complexity of the development process. Whether an organization is exclusively a consumer of components, a builder of components, or both, all aspects of their software development lifecycle are affected.

5.1 COMPONENT CONSUMER ISSUES

Let us first consider the component consumer and, for the sake of discussion, assume that useful components are readily available. Project management of component-based initiatives involves new challenges. Projects need to integrate a component culture throughout the software development life cycle in requirements specification, design, programming, testing and maintenance. A third party may control the life cycle of some of a project's components.

In requirements specification, the scope becomes broader, as developers need familiarity with not only their requirements, but also what components are available and the interoperability frameworks on which they depend. Components may work with one framework but not another.

In design, developers must choose whether to adapt the design to accommodate an off-the-shelf component, develop software to adapt the component to the requirement at hand, or forego the use of a component and just write code. Developing software to adapt the component may not be cost effective for a single instance, but if the component is useful in other, sufficiently numerous, circumstances, the adaptation cost may be justified. Such a judgment requires thorough knowledge of the requirements gained through domain analysis, which itself represents a new discipline for many, and will require learning.

Maintenance becomes more involved because of the myriad interactions. Requirements may change in one instance of component use, but not in others. A change in one component may require changes in interfaces to software modules and/or other components and the resulting testing. Components need to be certified reliable and efficient to save a consumer's effort. Component providers need to demonstrate that they or a third party have certified the component. Unless the consumer can trust the component's reliability, the consumer must perform certification. When components are certified, testing is limited to how they are configured and integrated and excludes their inner workings.

For the sake of our discussion above, we assumed components were readily available. However, components may not be readily available for many development projects—from either producers or external vendors.

5.2 COMPONENT PRODUCER ISSUES

Component producers must decide which components to produce. Producers need domain analysis to assess a component's efficacy. Rather than satisfying a requirement from a single consumer's perspective, the component producer attempts to divine a utility with the prospect of repeated use by, perhaps, several consumers in many similar requirement situations. Choices must be made about the level of granularity and interoperability.

Specialized components with complex functionality may limit the number of potential consumers. Fewer consumers across which to amortize the development cost means higher per consumer cost to acquire a component. Higher component acquisition cost eats into the potential life cycle savings of choosing a component over build-from-scratch, and so on. Component function granularity is a delicate design parameter for the component producer.

Consumer access to a component depends on which interoperability framework the producer chooses for it. The producer must follow the framework market and assess not only the current capabilities, but also the future capabilities that each framework vendor may be contemplating. These decisions must be revisited each time a component is revised or upgraded.

The testing task of component producers is difficult. Although components may have compact and well-defined interfaces, testing must be extremely thorough and perfectly consistent with the component documentation. A major determinant in the component consumer's adoption decision lies in the prospect of cost savings attributable to the component's reliability. From the developer's perspective this is a major challenge since testing must often occur independent of the eventual installation. Although not commonplace at this writing, third party certification of compatibility and reliability of components is inevitable as the component market evolves.

Documentation extends to configuration and to the domain, which increases the scope and expense of the documentation process. The wider the range of users of the documentation, the greater the importance and the required sophistication of the documentation. Components must be cataloged in a way that enables consumers to find and evaluate them.

5.3 SHARED PRODUCER / CONSUMER ISSUES

To achieve a significantly reduced time-to-market, a great deal of development must take place in parallel, requiring coordination among groups. While a component-based project may produce fewer lines of original programming, it must address an expanded number of management issues. Reducing the effort expended in traditional development with component reuse requires disciplined and effective management of the development, selection and use of components. Key to that is cost accounting.

Cost accounting is difficult for all IS organizations, and many ignore it or use only rough heuristics. Component reuse introduces additional project costs, including warehousing, delivery and use costs. Those costs affect each project that uses the component and raise the required initial investment. Any cost savings from reuse accrues back to the component.
From the purchase of packaged software through full service outsourcing of entire departments, outsourcing is a routine part of many IS organizations. Similarly, when deciding whether to build or buy components, management must determine if the savings from reuse will exceed the incremental cost of obtaining a reusable component. Typically, savings occur only when a component is reused several times. These reuses often occur over several years, introducing significant time value of money issues. In order to be cost effective, organizations must obtain reusable components that maximize the opportunity for repeated instances of reuse and limit the need for writing code.

If the organization finds that acquiring a component is the best option, it must still choose where to buy it. Vendor selection processes involve significant investment on the part of the purchasing organization, and the selling organization. The benefits of such screening are well established, but it is also important to make sure that the costs of the evaluation process do not become too high in the case of inexpensive components.

There are several viable pricing models for components. Components can be sold to outside consumers, who can then incorporate them into their own systems. They can also be licensed under a variety of terms including time period, volume, per application and per execution. Given that components can run on remote servers owned by the developer, licensing based on usage may become more common.

6.0 CONCLUSION

Components encompass many widely accepted traditional software engineering principles implemented with new technologies. The emergence of the standards and frameworks described in this chapter, along with the support of OMG, reflect the increasing maturity of the process of component development and use. The markets for components have been much slower to evolve. While there are many valuable generic components available, domain-specific components remain in short supply. Until more organizations start to use components, most domain-specific markets will be too small to support third-party development. Organizations considering wide-scale implementation of components will likely need to develop some components in-house.