# Part I. Introduction

# From Subroutines to Subsystems: Component-Based Software Development

Paul C. Clements

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, 15213, USA
pclement@sei.cmu.edu

## 1. Subroutines and Software Engineering

In the early days of programming, when machines were hard-wired and every byte of storage was precious, subroutines were invented to conserve memory. Their function was to allow programmers to execute code segments more than once, and under different (parameterized) circumstances, without having to duplicate that code in each physical location where it was needed. Software reuse was born. However, this was a different breed of reuse than we know today: This was reuse to serve the machine, to conserve mechanical resources. Reuse to save human resources was yet to come.

Soon, programmers observed that they could insert subroutines extracted from their previous programs, or even written by other programmers, and take advantage of the functionality without having to concern themselves with the details of coding. Generally-useful subroutines were collected into libraries, and soon very few people would ever again have to worry about how to implement, for example, a numerically-well-behaved double-precision cosine routine.

This phenomenon represented a powerful and fundamental paradigm shift in how we regarded software. Invoking a subroutine from a library became indistinguishable from writing any other statement that was built in to the programming language being used. Conceptually, this was a great unburdening. We viewed the subroutine as an atomic statement -- a *component* -- and could be blissfully unconcerned with its implementation, its development history, its storage management, and so forth.

Over the last few decades, most of what we now think of as software engineering blossomed into existence as a direct result of this phenomenon. In 1968, Edsger Dijkstra pointed out that how a program was structured was as important as making it produce the correct answer [2]. Teaching the principle of separation of concerns, Dijkstra showed that pieces of programs could be developed independently. Soon after, David Parnas introduced the concept of information-hiding [6] as the design discipline by which to divide a system into parts such that the whole system was easily changed by replacing any module with one satisfying the same interface. Design methodologists taught us how to craft our components so that they could live up to their promise. Prohibiting side effects, carefully specifying interfaces that guard implementation details, providing predictable behavior in the face of incorrect usage, and other design rules all contributed to components that could be plugged into existing systems. Object-oriented development was a direct, rather recent result of this trend.

## 2. Software engineering for components

Today, much of software engineering is still devoted to exploring and growing and applying this paradigm. *Software reuse* is about methods and techniques to enhance the reusability of software, including the management of repositories of components. *Domain engineering* is about finding commonalities among systems to identify components that can be applied to many systems, and to identify program families that are positioned to take fullest advantage of those components. *Software architecture* studies ways to structure systems so that they can be built from reusable components, evolved quickly, and analyzed reliably. Software architecture also concerns itself with the ways in which components are interconnected, so that we can move beyond the humble subroutine call as the primary mechanism for sending data to and initiating the execution of a component. Mechanisms from the process world, such as event signalling or time-based invocation, are examples. Some approaches can "wrap" stand-alone systems in software to make them behave as components, or wrap components to make them behave as stand-alone systems. The *open systems* community is working to produce and adopt standards so that components of a particular type (e.g., operating systems developed by different vendors) can be seamlessly interchanged. That community is also working on how to structure systems so they are positioned to take advantage of open standards (e.g., eschewing non-standard

Originally published in *American Programmer*, Vol. 8, No. 11, Nov. 1995, Published by Cutter Information Corp., Arlington, MA (617) 648-1950, http://www.cutter.com. Reprinted with permission.

operating system features, which would make the system dependent on a single vendor's product). The emerging *design patterns* community is trying to codify solutions to recurring application problems, a precursor for producing general components that implement those solutions.

## 3. CBSD: Buy, Don't Build

This paradigm has now been anointed with the name "Component-based software development" (CBSD). CBSD is changing the way large software systems are developed. CBSD embodies the "buy, don't build" philosophy espoused by Fred Brooks [1] and others. In the same way that early subroutines liberated the programmer from thinking about details, CBSD shifts the emphasis from *programming software* to *composing software systems*. Implementation has given way to integration as the focus. At its foundation is the assumption that there is sufficient commonality in many large software systems to justify developing reusable components to exploit and satisfy that commonality.

## 4. What's New?

In some ways, there is little new about CBSD; it is just a re-iteration of decades-old ideas coming to fruition. There are, however, some exciting new aspects.

**Increasing component size and complexity.** Today, available off-the-shelf components occupy a wide range of functionality. They include operating systems, compilers, network managers, database systems, CASE tools, and domain-specific varieties such as aircraft navigation algorithms, or banking system transaction handlers. As they grow in functionality, so does the challenge to make them generally useful across a broad variety of systems. Math subroutines are conceptually simple; they produce a result that is an easily-specified function of their inputs. Even databases, which can have breathtakingly complex implementations, have conceptually simple functionality: data goes in, and data comes out via any of several well-understood search or composition strategies. This conceptual simplicity leads to interface simplicity, making such components easy to integrate with existing software. But what if the component has many interfaces, with information flowing across each one that cannot be simply described? What if, for example, the component is an avionics system for a warplane that takes input from a myriad of sensors and manages the aircraft's flight controls, weapons systems, and navigation displays? From one point of view, this software is a stand-alone system; however, from the point of view of, say, an air battle simulator, the avionics software for each of the participating aircraft is just a component. The simulator must stimulate the avionics with simulated sensor readings, and absorb its flight control and weapons commands in order to represent the behavior of the aircraft in the overall simulation. Is it possible to make a plug-in component from such a complex entity? The Department of Defense is working on standards for just such a purpose, to make sure that simulators developed completely independently can interoperate with each other in massive new distributed simulation programs, in which the individual vehicle simulators are simply plug-in components.

**Coordination among components.** Classically, components are plugged into a skeletal software infrastructure that invokes each component appropriately and handles communication and coordination among components. Recently, however, the coordination infrastructure itself is being acknowledged as a component that is potentially available in pre-packaged form. David Garlan and Mary Shaw have laid the groundwork for studying these infrastructures in their work that catalogues *architectural styles* [3]. An architectural style is determined by a set of component types (such as a data repository or a component that computes a mathematical function), a topological layout of these components indicating their interrelationships, and a set of interaction mechanisms (e.g., subroutine call, event-subscriber blackboard) that determine how they coordinate. The Common Object Request Broker Architecture (CORBA) is an embodiment of one such style, complete with software that implements the coordination infrastructure, and standards that define what components can be plugged into it.

**Nontechnical issues.** Organizations are discovering that more than technical issues must be solved in order to make a CBSD approach work. While the right architecture (roughly speaking, a system structure and allocation of functionality to components) is critical, there are also organizational, process, and economic and marketing issues that must be addressed before CBSD is a viable approach. Personnel issues include deciding on the best training, and shifting the expertise in the work force from implementation to integration and domain knowledge. For organizations building reusable components for sale, customer interaction is quite different than when building one-at-a-time customized systems. It is to the organization's advantage if the component that the customer needs is most like the component the organization has on the shelf. This suggests a different style of negotiation. Also, customers can form user groups to collectively drive the organization to evolve their components in a particular direction, and the organization must be able to deal effectively with and be responsive to such groups. The organization must structure itself to efficiently produce the reusable components, while still being able to offer variations to important customers. And the organization must stay productive while it is first developing the reusable components. Finally, there are a host of legal issues that are beyond the scope of this paper and beyond the imagination (let alone the expertise) of the author.

## 5. Buying or Selling?

Different organizations may view CBSD from different viewpoints. A single organization might be a component supplier, a component consumer, or both. The combination case arises when an organization consumes components in order to produce a product that is but a component in some larger system.

Suppose an organization is producing a *product line*, which is a family of related systems positioned to take advantage of a market niche via reusable production assets. In this case, one part of the organization might be producing components that are generic (generally useful) across all members of the product line; the organization may be buying some of the components from outside vendors. Other parts of the organization integrate the components into different products, adapting them if necessary to meet the needs of specific customers. From a component vendor's point of view, product line development is often a viable approach to CBSD because it amortizes the cost of the components (whether purchased or developed internally) across more than one system.

## 6. Structuring a System to Accept Components

From a consumer's perspective, CBSD requires a planned and disciplined approach to the architecture of the system being built. Purchasing components at random will result in a collection of mis-matched parts that will have no hope of working in unison. Even a carefully-considered set of components may be unlikely to successfully operate with each other, as David Garlan has pointed out in his paper on architectural mis-match [4. The reason is that designers of software components make assumptions that are often subtle and undocumented about the ways in which the components will interact with other components, or the expectations about services or behaviors of those other components. These assumptions are embodied in the designs. Specific and precise interface specifications can attack this problem, but are hard to produce for complicated components. Still harder is achieving consensus on an interface that applies across an entire set of components built by different suppliers.

An architectural approach to building systems that are positioned to take advantage of the CBSD approach is the layered system. Software components are divided into groups (layers) based on the separation of concerns principle. Some components that are conceptually "close" to the underlying computing platform (i.e., would have to be replaced if the computer were switched) form the lowest layer. However, these components are required to be independent of the particular application. Conversely, components that are application-sensitive (i.e., would have to be switched if the details of the application requirements changed) constitute another layer. These components are not allowed to be sensitive to the underlying computing or communications platform. Other components occupy different layers depending on whether they are more closely tied to the computing infrastructure or the details of the application. The unifying principle of the layered approach is that a component at a particular layer is allowed to make use only of components at the same or next lower layer. Thus, components at each layer are insulated from change when components at distant layers are replaced or modified.

Figure 1 is an example of a layered scheme proposed by Patricia Oberndorf, an open systems expert at the Software Engineering Institute. In this scheme, computer-specific software components compose the lowest layer and are independent of the application domain. Above that lie components that would be generally useful across most application domains. Above that are components belonging to domains related to the application being built. Above that are components specific to the domain at hand, and finally special-purpose components for the system being built.

For example, suppose the system being built is the avionics software for the F-22 fighter aircraft. The domain is avionics software. Related domains are real-time systems, embedded systems, and human-in-the-loop systems. Figure 1 shows components that might reside at each layer in the diagram.

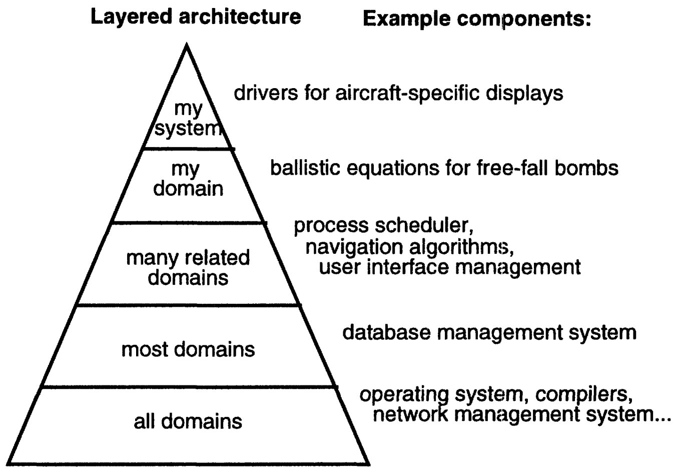
Layered architecture
Example components:
my system
drivers for aircraft-specific displays
my domain
ballistic equations for free-fall bombs
many related domains
process scheduler, navigation algorithms, user interface management
most domains
database management system
all domains
operating system, compilers, network management system...

**Figure 1. A domain-sensitive layered software architecture**

The triangle reflects the relative abundance or scarcity of components at each level. A system developer should not expect to find many components that exist that are specific to the system under construction. It will be easier to find and choose from components that are less domain-specific. For mid-level components, adopting data format and data interchange standards may aid in the search for components that can interoperate with each other.

Domain analysis techniques such as Feature-Oriented Domain Analysis (FODA) [5] can be of assistance in identifying the domain of the system, identifying related domains, and understanding the commonality and variation among programs in the domain of interest.

## 7. The Payoff and the Pitfalls

The potential advantages to successful CBSD are compelling. They include

- **Reduced development time.** It takes a lot less time to buy a component than it does to design it, code it, test it,debug it, and document it -- assuming that the search for a suitable component does not consume inordinate time.
- **Increased reliability of systems.** An off-the-shelf component will have been used in many other systems, and should therefore have had more bugs shaken out of it -- unless you happen to be an early customer, or the supplier of the component has low quality standards.
- **Increased flexibility.** Positioning a system to accommodate off-the-shelf components means that the system has been built to be immune from the details of the implementation of those components. This in turn means that any component satisfying the requirements will do the job, so there are more components from which to choose, which means that competitive market forces should drive the price down -- unless your system occupies a market too small to attract the attention of competing suppliers, or there has been no consensus reached on a common interface for those components.

Obviously, the road to CBSD success features a few deep potholes. Consider the questions that a consumer must face when building a system from off-the-shelf components:

- If the primary supplier goes out of business or stops making the component, will others step in to fill the gap?
- What happens if the vendor stops supporting the current version of the component, and the new versions are incompatible with the old?
- If the system demands high reliability or high availability, how can the consumer be sure that the component will allow the satisfaction of those requirements?

These and other concerns make CBSD a trap for the naive developer. It requires careful preparation and planning to achieve success. Interface standards, open architectures, market analysis, personnel issues, and organizational concerns all must be addressed. However, the benefits of CBSD are real and are being demonstrated on real projects of significant size. CBSD may be the most important paradigm shift in software development in decades -- or at least since the invention of the subroutine.

## 8. References

[1] Brooks, F. P. Jr., "No Silver Bullet: Essence and Accidents of Software Engineerig," *Computer*, vol. 20, no. 4, pp. 10-19, April 1987.

[2] Dijkstra, E. W.; "The structure of the 'T.H.E.' multiprogramming system," *CACM*, vol. 11, no. 5, pp. 453-457, May 1968.

[3] Garlan, D., and Shaw, M.; "An introduction to software architecture," in *Advances in Software Engineering and Knowledge Engineering,* vol. I, World Scientific Publishing Company,1993.

[4] Garlan, D., R. Allen, and J. Ockerbloom; "Architectural Mismatch (Why its hard to build systems out of existing parts)", *Proceedings, International Conference on Software Engineering*, Seattle, April 1995.

[5] K. Kang, S. Cohen, J. Hess, R. Novak, and S. Peterson; *Feature-Oriented Domain Analysis Feasibility Study: Interim Report*; technical report CMU/SEI-90-TR-21 ESD-90-TR-222, August 1990.

[6] Parnas, D.; "On the criteria for decomposing systems into modules," *CACM*, vol. 15, no. 12, pp. 1053-1058, December 1972.

# Engineering of Component-Based Systems

Alan W. Brown & Kurt C. Wallnau
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, 15213, USA
{awb, kcw}@sei.cmu.edu

## Abstract

*Many organizations are moving toward greater use of commercial off-the-shelf (COTS) components in the development of application systems. The development of component-based systems introduces fundamental changes in the way systems are acquired, integrated, deployed and evolved. In this paper we present a reference model for the assembly of component-based systems that can be used as the basis for defining a systematic approach to the development of such systems. This reference model is described, and used to explore a number of fundamental development issues.*

## 1. Introduction

For more than a decade good software development practice has been based on a "divide and conquer" approach to software design and implementation. Whether they are called "modules", "packages", "units", or "computer software configuration items", the approach has been to decompose a software system into manageable components based on maximizing cohesion within a component and minimizing coupling among components [13].

However, recently there has been a renaissance in the component-based approach to software development spurred on by two recent advances;

- the object-oriented development approach which is based on the development of an application system through the extension of existing libraries of self-contained operating units;
- the economic reality that large-scale software development must take greater advantage of existing commercial software, reducing the amount of new code that is required for each application system.

Both of these advances, object-oriented development and greater use of commercial off-the-shelf (COTS) software, raise the profile of software components as the basic building blocks of a software system. The development and maintenance of component-based systems, however, introduces fundamental changes in the way systems are acquired, integrated, deployed and evolved. Rather than the classic waterfall approach to software development, systems are designed by examining existing components to see how they meet the system requirements. This is followed by an iterative process of refining the requirements to match the existing components, and deciding how those components can best be integrated to provide the necessary functionality. Finally, the system is engineered by assembling the selected components using locally-developed code.

While many organizations are attempting to understand and take advantage of component-based software development, they lack a basic conceptual framework in which they can describe and discuss their needs, understand different methods and tools, and express issues and concerns. Without this conceptual framework confusion and misunderstandings often occur among engineers and managers, and decisions made cannot easily be rationalized or understood. A number of existing papers (e.g., [11][24]) describe various problems associated with the use of component-based systems, but without presenting a conceptual framework for organizing and categorizing the issues that must be addressed. Such a conceptual framework is a foundation on which further analyses and investigations can take place.

In this paper we present a reference model that can be used as the basis for defining a systematic approach to the development of component-based systems. The model is deliberately simple in its form in order to enable its use as a communication vehicle within and among different organizations. Despite the fact that the model concentrates primarily on technical aspects of the component assembly process and does not address many business and economic issues, we are finding that the model is invaluable as a means of describing and communicating between managers and engineers, for relating and focusing formerly disparate work, and as the basis for planning future tasks.

The remainder of this paper is organized as follows. Section 2 discusses the scope of component-based systems as they form the domain of our work in general, and this paper in particular. Section 3 describes the stages of the reference model in detail. Section 4 focuses on the transitions between states in the reference model and uses this to explore a number of fundamental issues with respect to the development of component-based systems. Section 5 summarizes the paper, and points to future work that is required to expand the model and its application.

## 2. Component-based Software Development

While all (real) systems are composed of components, in our usage *component-based* systems are comprised of multiple software components that:

- are ready "off-the-shelf," whether from a commercial source (COTS) or re-used from another system;
- have significant aggregate functionality and complexity;
- are self-contained and possibly execute independently;
- will be used "as is" rather than modified;
- must be integrated with other components to achieve required system functionality.

Examples of component-based systems can be drawn from many domains, including: computer-aided software engineering (CASE), design engineering (CADE) and manufacturing engineering (CAME); office automation; workflow management; command and control, and many others.

From a component-based perspective the process of system design involves the selection of components, together with an analysis of which components can be acquired from external sources (e.g., COTS) and which ones must be developed from scratch. In contrast to the development of other kinds of systems where system integration is often the tail-end of an implementation effort, in component-based systems determining how to integrate components is often the primary task performed by designers. As a result, component integration needs are vital to the component selection process, and a major consideration in the decision to acquire or build the components.

The importance of the integration process has been illustrated in a number of application domains. For example, in the CASE domain the study of component integration is well-established (e.g., see [3][2]). In this domain a number of conceptual models of the integration issues have been developed, and mechanisms specifically targeted at CASE tool integration have been produced. From this work it has been found that an understanding of the component integration process is vital to enable:

- component vendors to develop components that can more easily be integrated with others;
- component integrators to develop efficient engineering techniques that produce coherent systems with acceptable functional and afunctional properties;
- system acquirers to evaluate alternative system engineering solutions for their suitability within a given context;
- system users to understand the implications of requested changes and enhancements to operational component-based systems.

The key to the reference model that follows is a recognition of the central role played by integration in the engineering of component-based systems, and the help this recognition provides in understanding many of the challenges being faced in making component-based system development both efficient and effective.

## 3. A Reference Model for Component-Based Systems

As suggested in the previous section, the engineering of component-based systems can be considered to be primarily an assembly and integration process. This suggests a reference model for describing the engineering practices involved in assembling component-based systems, as depicted in Figure 1..

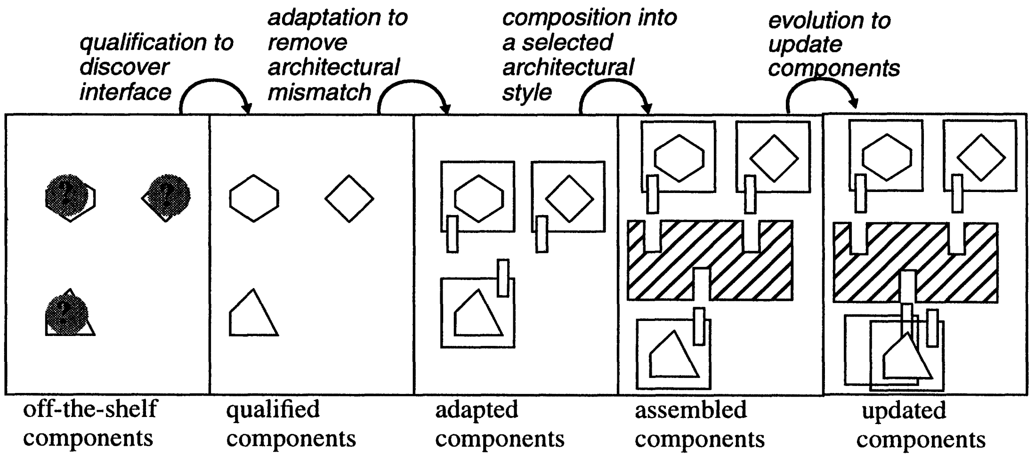
qualification to discover interface
adaptation to remove architectural mismatch
composition into a selected architectural style
evolution to update components
off-the-shelf components
qualified components
adapted components
assembled components
updated components

**Figure 1. A Reference Model For Architectural Assembly of Components**

The vertical partitions depicted in Figure 1. describe the central artifact of component-based systems—the components—in various states. Briefly stated, these partitions are as follows:

- *Off-the-shelf components* have hidden interfaces (using a definition of interface that encompasses all potential interactions among components, not just an application programming interface[18]).
- *Qualified components* have discovered interfaces so that possible sources of conflict and overlap have been identified. This is (by the definition of interface) a partial discovery: only those interfaces important to effective component assembly and evolution are identified.
- *Adapted components* have been amended to address potential sources of conflict. The figure implies a kind of component "wrapping," but other approaches are possible (e.g., the use of mediators and translators).
- *Assembled components* have been integrated into an architectural infrastructure. This infrastructure will support component assembly and coordination, and differentiates architectural assembly from ad hoc "glue."
- *Updated components* have been replaced by newer versions, or by different components with similar behavior and interfaces. Often this requires wrappers to be rewritten, and for well-defined component interfaces to reduce the extensive testing needed to ensure operation of unchanged components is not adversely effected.

Having briefly described the main panels of Figure 1, we now discuss each panel in more detail.

## 3.1 Off-the-shelf Components

Components come from a variety of sources, some developed in-house (perhaps used in a previous project), others specifically purchased from a commercial vendor (typically packages such as database management systems and network monitors). This variety of sources is both a major strength of the component-based approach, and a major challenge, since the components have varied pedigree, many unknown attributes, and are of varied quality.

Selection and evaluation of components are the key activities that take place early in the life cycle of a component-based system. Typically, once the broad functional requirements for a system are known, an organization makes an inventory of known local and external components that may provide elements of that functionality. How the organization proceeds from this point in evaluating the available components depends upon the individual organization.

Many organizations have relatively mature evaluation techniques for selecting from among a group of peer products, and this is reflected in the literature. For example, a number of papers describe general product evaluation criteria (e.g., [10]), while others describe specialized techniques which take into account the particular needs of specific application domains (e.g., CASE tools [9] or testing tools [19]). These evaluation approaches typically involve a combination of paper-based studies of the components, discussion with other users of those components, and hands-on benchmarking and prototyping. The complexity of component selection has already been recognized in software engineering literature—a decision framework that supports multi-variable component selection analysis has been developed in order to address some of this complexity [11].

A recent trend in some organizations is toward a "product-line" approach to software development based on a reusable set of components that appear in a range of software products [15]. This approach is based on the premise that similar products (e.g., most radar systems) have a similar software architecture, and a majority of the required functionality is the same from one product to the next. The common functionality can be provided by the same set of components, simplifying the development and maintenance life-cycle.

## 3.2 Qualified Components

Typically, a component is described in terms of an interface that provides access to component functionality. In many cases this interface is the main source of information about a component: the user manual is often simply a step-by-step guide to the operations available at this interface. However, in a much broader interpretation the interface to a component includes far more than the functionality it provides. To make use of a component one must also understand aspects of the components performance, reliability, usability, and so on.

As a result, for most components there are many unknowns. While much information about a component and its operation can be found through hands-on evaluation, a number of gaps will remain. For example, in real-time safety-critical domains certification of software systems relies on gaining information about mean time between failures, performing timing analysis using techniques such as rate monotonic analysis, and bounding resource usage by ensuring that livelocks and deadlocks cannot occur. In assembling systems from components information on each component is required to complete the analyses. This information includes failure rates, performance models, and detailed software design data. Such information is rarely provided with COTS components, and is frequently missing from in-house component libraries.

There is little established work on how to carry out qualification of software components. In fact, there is little agreement on which attributes of a component are critical to its use in a component-based system. Perhaps the most useful recent work that begins to address this issue is the work on describing quality attributes of software systems. For example, Bass et al. [1] have developed a software architecture analysis method (SAAM) that distinguishes between intrinsic and extrinsic properties of a component, and investigates the influences of those properties on a software architecture. However, much further work remains in this area.

Qualification of a component can also extend to a qualification of the development process used to create and maintain a component. Again, this is most obvious in safety-critical applications (for example, ensuring algorithms have been validated, and that rigorous code inspections have taken place). However, the concept is extending beyond the safety-critical domain. For example, many organizations now insist that contractors are ISO 9000 certified [20], or that they have reached a certain level on the SEI's Capability Maturity Model (CMM) [12]. This ensures that the software components they produce have been developed using well-defined practices and procedures.

## 3.3 Adapted Components

The variety of sources for components leads to a number of problems. These arise because stand-alone components are being used to construct a system where the components must cooperate. The result is a number of conflicts with respect to concerns such as:

- sharing resources such as memory, swap space, and printers;
- carrying out common activities such as data management, screen management, and version control;
- environment set-up using environment variables, temporary files, and common naming policies.

These conflicts are symptoms of a more pervasive problem in engineering component-based systems: architectural mismatch of components [7]. That is, the wide variety of conflicting operating assumptions made by each component. (This issue is examined in much more detail in the next section.)

As a consequence of these conflicts, components must be adapted based on a common notion of component "citizenship" (i.e., based on rules that ensure conflicts among components are minimized). This usually involves some form of component wrapping — locally developed code that provides an encapsulation of the component to mask unwanted and incompatible behavior. Different forms of wrapping exist based on how much access is provided to the internal structure of a component. The approaches can be classified as white box, where access to source code allows a component to be significantly re-written to operate with other components, grey box, where source code of a component is not modified but the component provides its own extension language or application programming interface (API), and black box, where only a binary executable form of the component is available and there is no extension language or API [23].

Wrapping is not the only approach to overcoming component incompatibilities. Another approach is through the use of mediators. Informally, we can view a mediator as an active agent that coordinates between different interpretations of the same system property. For example, a mediator may translate between different data formats, establish common events, or define common administrative policies (e.g., for security and access control).

## 3.4 Assembled Components

The assembled components are integrated through some well-defined infrastructure. This infrastructure provides the binding that forms a system from the disparate components.

It is useful to consider at least three different levels at which the infrastructure must operate:

- At the highest abstract level the infrastructure embodies a coordination model that defines how the different components will interact to carry out the required end-user functionality. It is the role of the infrastructure to allow this coordination model to be readily described, validated, and enacted.
- At a lower level the infrastructure provides services that will be used by the components to interact and to carry out common tasks. The interface to these services must be complete and consistent.
- At the most practical level the infrastructure is itself a software component that implements the necessary coordination services required. It must be well-written so that it is easily understood, perform effectively, and be readily updated to new modes of component interaction.

There is a long history of developing infrastructure capabilities based on three classes of technology: operating systems, database management systems, and messaging systems. Each has their particular strengths and weaknesses. Currently, most active research is taking place with the use of messaging systems as infrastructure providers, particularly using object request brokers (ORBs) conforming to the Common Object Request Broker Architecture (CORBA). While use of ORBs is immature, initial reports cite a number of advantages to their use in building component-based systems [25][17].

## 3.5 Updated Components

As with any system, a component-based system must evolve over time to fix errors and to add new functionality. Here again, component-based systems bring their own strengths and weaknesses.

On the surface a component-based approach brings advantages in terms of its ease of evolution: components are the unit of change. Hence, to repair an error an updated component is swapped for its defective equivalent treating components as plug-replaceable units. Similarly, when additional functionality is required it is embodied in a new component which is added to the system.

However, this is too simplistic a view of component upgrade. Replacement of one component with another is often a time-consuming and arduous task since the new component must be thoroughly tested in isolation and in combination with the rest of the system. Wrappers must typically be re-written, and side effects from changes must be found and assessed.

Compounding this upgrade problem is that component producers frequently upgrade their components based on error reports, perceived market needs, and product aesthetics. New component releases require a decision from the component-based system developer on whether or not to include the new component in the system. To answer" yes" implies facing an unknown amount of re-writing of wrapper code and system testing. To answer "no" implies relying on older versions of components that may be behind the current state-of-the-art and may not be adequately supported by the component supplier.

As an illustration of the impact of component upgrades consider a system consisting of 12 COTS components, each of which is released as a new version every 6 months. To keep up with the latest version of each component requires on average a system upgrade every two weeks. If new releases are not installed, analysis of which component versions are compatible become more and more difficult, leading to a system administrator's nightmare.

In safety-critical systems the problems of system upgrades are even more acute. Not only is the testing and analysis of new components a critical yet time-consuming endeavor, it is usually unacceptable to have long periods of time in which the system is unavailable. In such cases complex techniques for dependable upgrade of highly-available systems must be employed (e.g., using approaches such as the Simplex architecture [21]).

## 3.6 Summary

In this section a reference model for the engineering of component-based systems has been described. The model highlights the component assembly process and the various activities that must take place in building a component-based system.

In the next section we make use of the reference model as a framework for describing a number of issues that distinguish the engineering of component-based systems from traditional development approaches.

## 4. Component Integration Issues

Using the concepts of the reference model presented in Section 2, we describe what we believe are four key issues that must be addressed in the engineering of component-based systems. We begin with a discussion of system life-cycle issues peculiar to component-based systems, then focus more narrowly on the technical issues that arise as a consequence of transitioning between key states (panels) of the architectural assembly reference model depicted in Figure 1.

### 4.1 Understanding the Component-Based Systems Life-cycle

The traditional software engineering life-cycle is not applicable when engineering systems from components. The typical waterfall phases of requirements, design, implementation, test, and maintenance clearly still apply. However, due to extensive use of existing components, the activities involved in each phase and the relationships among phases are often significantly changed from current approaches.

We can characterize the component-based approach as essentially one of negotiating a set of engineering trade-offs for the integration of a set of existing components to satisfy a particular set of requirements. The key to the engineering of component-based systems is to understand what trade-offs are being made, to record the rationale used in making trade-off decisions, and to evaluate how those trade-offs affect the resultant product in practice. This provides the necessary basis for improving the system as requirements or operating conditions evolve over time.

Here we consider 2 key engineering trade-offs involved in assembling and evolving component-based systems.

*System requirements vs. available components*

The initial stages of developing a component-based system involve the selection of components that are likely to satisfy the main system requirements. However, most organizations restrict selection of components to a small number in recognition that the organization can be more efficient if it does not manage too diverse a set of components. Thus, scanning the complete marketplace for every possible component that may be of interest is typically not carried out: an organization uses those components with which it is familiar.

As a result of initial component selection, the typical "80/20" rule often applies — 80% of the system functionality can be provided relatively easily with the selected components, while the remaining 20% can only be provided with much more difficulty. At this point the system engineers may consider what changes to the system requirements may make the use of selected components more effective. Once identified, these can be discussed with the users of the system to understand the priority of these requirements, and make decisions concerning which requirements can be amended.[1]

1. In fact, recently proposed changes in U.S. DoD policy state that where greater use of COTS components is possible, the system requirements should be changed to facilitate this.

In component-based system development, system designers must address a number of important buy versus build decisions with respect to many parts of the system. Such decisions are faced in all application domains, but are particularly important in safety-critical systems where reliability, availability, and predictability are essential. Often the majority of the system is developed in-house to ensure the organization has complete understanding and control of the system operation and evolution. In spite of any savings that may accrue, using commercial components in such situations is frequently considered too high a risk.

*System architecture vs. component interfaces*

The architecture of a system defines the basic components of the system and their connections [8]. Many alternatives exist in designing the architecture of a system, each with its own strengths and weaknesses in terms of how that architecture provides the key attributes of the system. A primary task of a system engineer is to evaluate the architectural alternatives by prioritizing system attributes based on perceived end-user needs. Components must then be assembled to provide those attributes.

Such an approach raises a number of challenges. First, it is unclear what are the attributes of a system that most contribute to a system's efficiency and effectiveness, and how such attributes can practically be assessed. Second, the relationships among afunctional attributes (e.g., performance, dependability, usability, maintainability, etc.) is poorly understood. While practical experience has identified a number of trade-offs between afunctional attributes of a system (e.g., between performance and maintainability), analytical techniques for examining these trade-offs are still lacking. Finally, it has been found in practice that there are characteristics of a component interface that facilitate some kinds of system architecture while precluding others [16]. For example, the granularity of access to component data often influences the rate at which data synchronization among components can practically occur. Further work is required to more fully understand how component interfaces influence the component assembly process.

While attempts have taken place to address these challenges by classifying system architectures in terms of their key attributes, currently system engineers primarily base their system architectures on past experience in building similar systems. Particular models of system behavior are reused from one system to another, modifying the architecture to meet the particular characteristics of the specific components being used.

## 4.2 Interface Discovery and Analysis

Obviously, a pre-condition for successful integration of components is that their interfaces are known. Borrowing a phrase from the hardware domain (always a risky proposition in the software domain), component integrators need to discover the *function and form* of software components—the services provided, and the means by which consumers access these services, respectively.

Different degrees of sophistication can be found in techniques used to discover the form and function of software components. At one extreme a few key functions may be of such high priority as to obviate the need for exhaustive component analysis. The discovery process in these cases can be as simple as browsing vender literature, examining programmer documentation, etc. At the other extreme component classes (e.g., database, geographic information system, network management, spreadsheet) can be modeled in terms of features. Then, particular components can be described via functional profiles against these component-domain models, much like *Consumer Reports* might describe different brands of televisions or food processors. This more robust discovery approach has been adopted by COTS-focused efforts that espouse a product-line approach to component-based systems [4].

However, even a complete understanding of component functionality and the interfaces to this functionality is insufficient for anything but trivial system integration problems. In fact, the *complete* interface of a component includes more than just the mechanisms that a component uses to make its functionality available to clients: it includes all of the assumptions made by a component about integration-time and run-time uses of the component. For example, each of the following might be considered part of a component interface:

- application programming interface (API);
- required development and integration tools;
- secondary storage requirements (run-time), processor requirements (performance) and network requirements (capacity);
- required software services (operating system, or from other components);
- security assumptions (access control, user roles, authentication);
- embedded design assumptions, such as the use of specific polling techniques;
- exception detection and processing.

This list—which is by no means exhaustive—illustrates that implementation decisions normally thought to be "hidden" by abstract interfaces play a crucial role in determining whether, and how easily, components can be integrated. Further, these assumptions are not easily detected from vendor literature or functional interface specifications.

To this list we can also add an additional class of component properties sometimes referred to as "quality attributes" [10]. Assumptions about the run-time environment may impinge upon quality attributes such as reliability; for example, a component may not be designed for continuous operation, or operation in environments that may experience varying degrees of degraded performance.

The essential problem we are describing is that our current interface specification techniques do not adequately address all of the properties exhibited by components that will determine, ultimately, the integrability of the component. Nor do we yet know all of the different kinds of assumptions that components can make that may result in architectural mismatch. Thus, we are hampered from discovering the interface of a component because we are not always sure what we need to look for. Worse, we do not have well-defined notations or theories for describing and measuring all interface properties, especially those relating to quality attributes. All of this is compounded, of course, when we are confronted with components that are "black boxes" as is the case with COTS components: we are groping in the dark to discover the features of a component that lies on the other side of a locked door.

### 4.3 Removal of Architectural Mismatches

Assuming that we can discover the *complete* interface of a component, the next step is to repair any mismatches that have been detected among components[2]. Given the difficulty of discovering complete interfaces, however, this would seem to be a glib prescription.

At this juncture the role of *software architecture* can be asserted. Software architecture deals with high-level design patterns, often expressed as components, connectors and coordination [8]. Components[3] refer to units of functionality, connectors with the integration of components, and coordination as the manner in which components interact at run-time. One role of software architecture (in component-based systems) is to restrict the classes of potential mismatches that might arise among component interfaces, and thus offer useful constraints for the interface discovery process. For example, by prescribing a specific coordination model, a software architecture in effect identifies and prioritizes the *key* run-time interfaces a component must exhibit either natively or via adaptation.

The topic of component adaptation has received even less attention than interface discovery. The consensus seems to be that this process is inherently ad hoc, low-level and very messy. Various euphemisms for adaptation such as "glue" and "chewing gum" are revealing in themselves. A euphemism with more sanitary connotations is "wrapper." Nevertheless, considering the important role that component adaptation code will play in large-scale component-based systems—especially those with safety or security requirements—it seems necessary to gain a firmer grasp of this topic.

2. Many of the latent "bugs" in component-based systems arise because of undetected, and hence un-repaired, architectural mismatch.

3. The component-based definition of this term is more restrictive than its use in software architecture literature.

A first step along this road is to classify the different adaptation techniques that are possible, and to understand when their uses are required. For example, glue and wrapper are evocative of different component adaptation approaches. Wrappers suggest that a component is adapted by encasing it within a virtual component that presents an alternative, translated interface; glue suggests a somewhat less contained approach, with code oozed between components, for example a collection of shell scripts and filters.

More generally, integration involves a relationship between entities[22]. Adaptation can occur at one or all endpoints of a relationship, or on the relationship itself, for example through the introduction of an intermediary component (a.k.a. "mediator"). The software architecture may provide mechanisms to facilitate component adaptation; alternatively, adaptation mechanisms may lie outside of the scope of the architecture. Examples of both have appeared in practice.

### 4.4 Architecture Selection & System Composition

The selection of a particular architectural style, or the invention of a custom style, is perhaps the most important design decision of all. The functionality of a component-based system will be found in the components; the quality attributes (e.g., security, maintainability) will be found in the architecture. Beyond this, the architecture will drive the integration effort: as discussed in the previous section, architecture defines the integration-time and run-time contexts into which components must be adapted.

For example, in developing a command and control system from COTS components it may be possible to select from a number of architectural styles:

- database, in which centralized control of all operation data is the key to all information sharing among components in the system;
- blackboard, in which data sharing among components is opportunistic involving reduced levels of system overhead;
- message bus, in which components have separate data stores coordinated through messages announcing changes among components.

Each architectural style has its own strengths and weaknesses in terms of overall system qualities, and requires different considerations in the selection of components.

Despite the importance of this early design decision, our understanding of architectural styles, the combination of different styles, and the use of one or more styles to achieve targeted quality attributes, is still very immature. Moreover, little work has been focused directly upon the question of which architectural styles are best-suited to component-based systems. To be sure, canonical architectural styles have emerged from application domains that exhibit component-based properties, e.g., integrated computer-aided software engineering (CASE) [3]. Nevertheless, the CASE experience has been a mixed success at best; in any event, different application domains have different needs and hence will require different architectural approaches than found in CASE.

To illustrate the still-unsettled nature of our understanding of architectures for component-based systems, consider the fundamental dichotomy between function-oriented and structure-oriented architectural approaches. The function-oriented approach is by far the predominant approach to component-based systems. It defines components to match available off-the-shelf components, and defines interfaces in terms of component functionality. Greater abstraction is sometimes obtained by generalizing component-specific interfaces, thus encapsulating some technology and vendor dependencies. A good illustration of a function-oriented approach has been defined in the command and control domain [14]. Function-oriented architectures are good for describing system functionality and for integrating specific functionality but are weak for addressing the run-time properties of a design, e.g., throughput, latency and reliability.

The structure-oriented approach has emerged as the study of software architecture has intensified. Rather than defining component interfaces in terms of functionality, structural styles define interfaces in terms of the role a component plays in a *coordination model*—a model that describes how the components interact. A simple illustration of a structural style is UNIX pipes and filters; more sophisticated illustrations include structural models for flight simulators [5], the Simplex architecture for evolvable dependable real-time systems [21], and prototype architectures for distributed workflow management [6] and distributed manufacturing design engineering [25]. The structural approach yields architectures that support analysis of dynamic system properties, but are not optimized to support access to component-specific functionality.

Which of these approaches should be adopted? What about approaches that merge some aspects of both—a structural approach to describe how components interoperate, with functional extensions as common infrastructure facilities? Such a hybrid is found in an emerging, standard high-level architecture for interoperable simulations [5]. Whatever approach is taken will have a profound effect on the integrability of a component-based system. In addition, better design heuristics are needed to select among these (and many other) architectural trade-offs.

### 4.5 Predictable Component Update

Systems in operational use must periodically be updated. Typically, this involves developing an updated system, performing extensive testing, and then switching over to the new system in the field. A number of potential problems arise in doing this:

- identifying and bounding the changes that are required to the system;
- testing the system sufficiently to ensure that the updates do not have undesirable consequences on the rest of the system;
- ensuring that the old system can be put back on-line should the behavior of the new system be unacceptable,

In mission-critical application domains (e.g., avionics or health care patient monitoring) the consequences of poorly planned and implemented system upgrades are sufficient that often systems are not upgraded. This occurs despite the availability of improved algorithms, techniques, and technology that could improve the system's overall effectiveness.

Engineering systems from components further complicates the upgrade problems, since many of the components are maintained by third party organizations. As a result, changes to those components are often not well-understood by system integrators and end-users, and often not documented in sufficient detail. This leaves system integrators with the challenge of evolving component-based systems in a predictable, dependable way in the face of incomplete information about the components to be upgraded.

A number of approaches have been taken to try to address these problems. One of the most interesting involves the design of a layer of software that supports reliable and safe upgrade of on-line systems [21]. This layer of software, known as the Simplex Architecture, is based on three key ideas: components as replaceable system units, controlled replacement transactions, and a real-time publish and subscription facility. The Simplex Architecture also supports the use of analytic redundancy through a fault-tolerant protocol known as the Simple Leadership Protocol (SLP). In addition to supporting system evolution, model-based voting used by SLP allows the tolerance of combined hardware and software failures.

A number of demonstrations of this approach have been built. In one of these based on a triplicated computer fault tolerant group utilizing SLP, it was shown that safe on-line upgrade of application software, operating systems, and hardware components are all feasible using this approach. While many issues remain to be addressed with this approach (e.g., scaleability issues when applied to large systems), it provides an very useful illustration of how dependable evolution of component-based systems may be possible in the future.

## 5. Summary and Conclusions

The trend towards greater reliance on off-the-shelf components for even the most complex software systems is increasingly clear. Numerous large system acquisitions in the US DoD and Government agencies in very demanding application domains—air traffic control, real-time simulators, and command and control to name just a few—are pushing the limits of our ability to develop systems with predictable properties.

Unfortunately, engineering practices have not been keeping pace with the changing nature of system building. Our ability to describe the key properties of software components to enable their rapid and error-free integration is inadequate; our techniques for discovering the interfaces of previously-developed, possibly COTS components is significantly hampered; and our understanding of architectural patterns best suited for component-based systems, and the techniques best suited for adapting components to these patterns is still quite immature.

However, much work is in progress to try to address these challenges. Interface discovery techniques are being developed by extracting experiences from system integrators; understanding of software architectures is maturing; and system evolution techniques are emerging which allow safe on-line upgrade of component-based systems.

In this paper we have explored the challenges to engineering of component-based systems by presenting a reference model of the key activities in the constructive phases of component-based systems development (assembly and re-assembly/evolution). Although we briefly touched on other aspects of component-based systems, such as requirements acquisition processes sensitive to off-the-shelf component reuse, our focus has been on the technical aspects of integrating component-based systems. The reference model we presented, while not complete or detailed, provides a good foundation for discussing key component integration issues and for relating different trends in software engineering (notably, software architecture and component-based systems).

*Acknowledgments*

We are grateful to David Carney and Ed Morris for their valuable comments on earlier drafts of this paper.

The SEI is sponsored by the U.S. Department of Defense.

## 6. References

[1] Abowd, G., Bass, L., Kazman, R., Webb, M., "SAAM: A Method for Analyzing the Properties of Software Architecture," in proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, pp. 81-90, May 1994.

[2] Brown, A.W. and Penedo, M.H., "An Annotated Bibliography of Software Engineering Environment Integration", ACM Software Engineering Notes V17 #3, pp47-55, July 1992.

[3] Brown, A.W., Carney, D.J., Morris, E.J., Smith, D.B., Zarella,P.F., *Principles of CASE Tool Integration*, Oxford University Press, 1994.

[4] Comprehensive Approach to Reusable Defense Software (CARDS) Home Page, http://www.cards.com

[5] Defense Modeling and Simulation Organization, High-Level Architecture, http://www.dmso.mil/project/hla

[6] Earl, A., Long, F., and Wallnau, K., "Towards a distributed, mediated architecture for workflow management," To appear in proceedings of NSF Workshop on Workflow Management: State of the Art and Beyond, Athens, GA, May 8-10, 1996.

[7] Garlan, D., Allen, R., Ockerbloom, J., "Architecture Mismatch: Why Reuse is so Hard", IEEE Software V12, #6, pp17-26, November 1995.

[8] Garlan, D. and Shaw, M., "An Introduction to Software Architecture," in Advances in Software Engineering and Knowledge Engineering, vol. I, World Scientific Publishing Company,1993

[9] IEEE Recommended Practice on the Selection and Evaluation of CASE Tools, P1209, 1994.

[10] Information Technology — Software Product Evaluation — Quality Characteristics and Guidelines for their Use, International Standards Organisation (ISO), ISO/IEC 9126:1991, 1991.

[11] Kontio, J., "A case study in applying a systematic method for COTS selection" proceedings of the 18th International Conference on Software Engineering (ICSE), pp. 201-209, March 1996.

[12] Paulk, M.C., Curtis, B., & Chrissis, M.B. *Capability Maturity Model for Software*. Technical Report CMU/SEI-91-TR-24, ADA240603, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, August 1991.

[13] Parnas, D.L., "On the criteria to be used in decomposing systems into modules," Communications of the ACM, Vol. 15, No. 2, pp. 1053-1058, 1972.

[14] PRISM Generic Command Center Architecture, http://www.cards.com/PRISM/prism_gcca.html

[15] MBSE Home Page, http://www.sei.cmu.edu/technology/mbse/mbse.html

[16] Nejmeh, B., *Characteristics of Integrable Tools*. Technical Report INTEG_S/W_TOOLS-89036-N Version 1.0, Software Productivity Consortium, May 1989.

[17] Orfali, R., Harkey D., and Edwards, J., "The Essential Distributed Objects Survival Guide", John Wiley and Sons, Inc., 1996.

[18] Parnas, D.L., "Information distribution aspects of design methodology," in proceedings of IFIP conference, 1971, North Holland Publishing Co.

[19] Poston R.M., and Sexton M.P., "Evaluating and Selecting Testing Tools", IEEE Software,V9, #3, pp33-42, May 1992.

[20] Schumauch C.H., *ISO 9000 for Software Developers*, ASQC Quality Press, 1994.

[21] Sha, L., Rajkumar, R., and Gagliardi, M., *A Software Architecture for Dependable and Evolvable Industrial Computing Systems*, Technical Report CMU/SEI-95-TR-005, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, July 1995.

[22] Thomas, I. and Nejmeh. B., "Definitions of Tool Integration for Environments" IEEE Software, Vol 9, No.3, pp. 29-35, March 1992.

[23] Valetto, G. and Kaiser, G.E., "Enveloping Sophisticated Tools into Computer-Aided Software Engineering Environments", Proceedings of 7th IEEE International Workshop on CASE, July 1995.

[24] Vidger, M.R., Gentleman, W.M., and Dean, J., "COTS Software Integration: State-of-the-art", Technical Report, National Research Council Canada, January 1996. http://wwwsel.iit.nrc.ca/abstracts/NRC39198.abs

[25] Wallnau, K., and Wallace, E., "A Robust Evaluation of the Object Management Architecture: A Focused Case Study in Legacy Systems Migration" submitted to OOPLSA'96.