From *The American Programmer*, vol. 8, no. 11, November 1995

From Subroutines to Subsystems: Component-Based Software Development [1]

Paul C. Clements
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213 USA

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 [Dijkstra 68]. Teaching the principle of separation of concerns, Dijk-

1. This work is sponsored by the U. S. Department of Defense.

stra showed that pieces of programs could be developed independently. Soon after, David Parnas introduced the concept of information-hiding [Parnas 72] 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.

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” standalone 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 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.

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

[Brooks 87] 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.

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* [Garlan 93]. 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.

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.

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 mismatch [Garlan 95]. 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 lay-

ers 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.

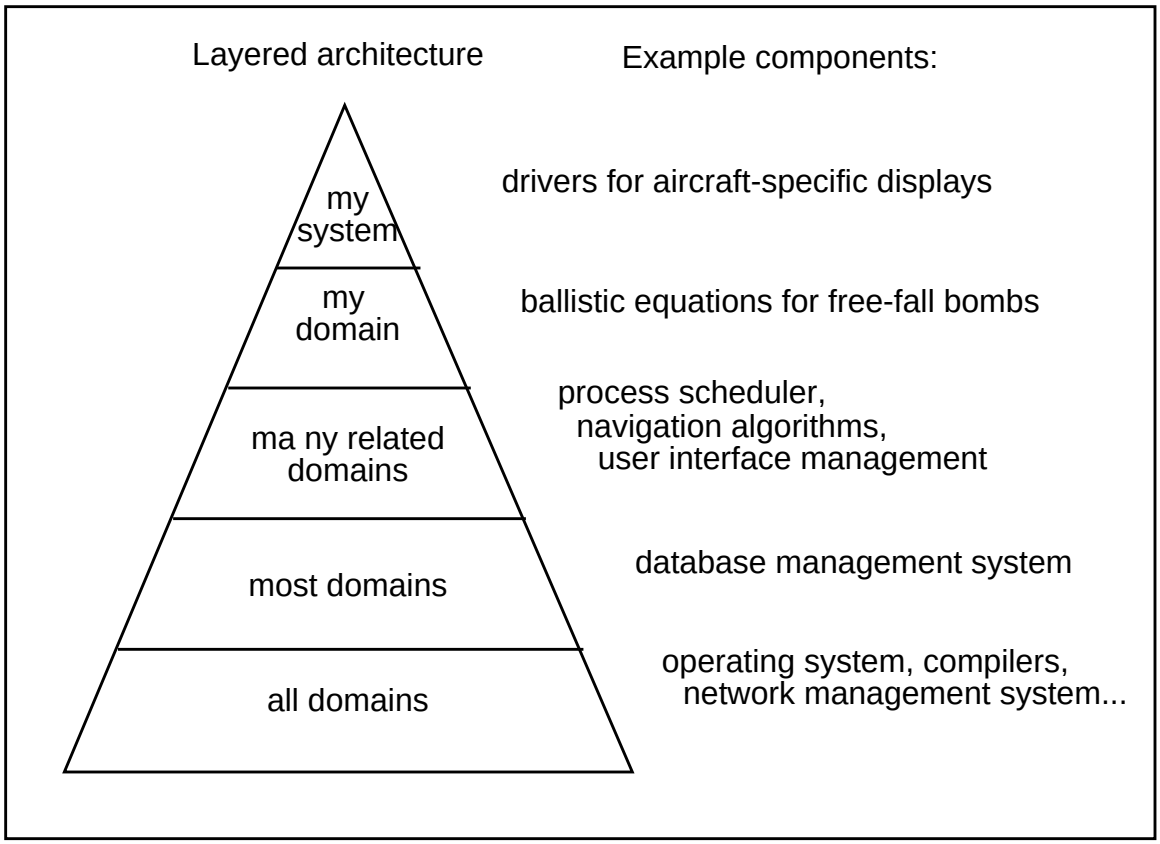

Figure 1: A domain-sensitive layered software architecture

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) [Kang 90] 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.

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.

References

Brooks 87 — Brooks, F. P. Jr., “No Silver Bullet: Essence and Accidents of Software Engineerig,” *Computer*, vol. 20, no. 4, pp. 10-19, April 1987.

Dijkstra 68 — Dijkstra, E. W.; “The structure of the ‘T.H.E.’ multiprogramming system,” *CACM*, vol. 11, no. 5, pp. 453-457, May 1968.

Garlan 93 — 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.

Garlan 95 — 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.

Kang 90 — 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.

Parnas 72 — Parnas, D.; “On the criteria for decomposing systems into modules,” *CACM*, vol. 15, no. 12, pp. 1053-1058, December 1972.