# Management of Component-Based Software Engineering

By
Shahzad (Shah) Bhatti, R&D Project Manager, Hewlett Packard

The advent of component-based software engineering provides managers with opportunities to streamline their software development process. These opportunities exist in all phases of the software development process, from analysis to maintenance, and from project planning to project tracking. This paper will identify specific areas of software engineering that may benefit from component-based software engineering.

Component-based software engineering by virtue of its inherent nature complies with the software engineering principles of modularity, abstraction, encapsulation and information hiding. The first step is to create an interface specification, which is a critical part of the analysis or design phase of component-based software engineering. An Interface Definition Language (IDL) is often used to create the interface code and documentation.

Several approaches are suitable for the creation of interface specification. A “team of architects” or a “convention of engineers” may create the interface specification. The “team of architects” approach will yield the interface specification most efficiently. On the other hand, the “convention of engineers” approach will provide a broad-based understanding and support of interfaces. The analysis and design phase must produce the interface specification and a scenario description of how the interfaces interact to accomplish specific tasks.

One question inevitably comes up when converting a conventional software system to a component-based software system: Should the system architecture be changed before converting it to components? The answer to this question is not simple. It depends on where you want to invest your resources. An area of the system, which is stable and not expected to change in the future, is a good candidate for conversion into a component without any investment in changing its architecture. However, an area of the system, which is unstable and dynamic may require architectural changes before conversion to components. There are some areas of the systems that may not be converted to components at all. For example, the boot code in an embedded system.

Another question, which often arises, is: In what order should the conventional system be converted to a component-based system? The best approach is to divide the system into logically related sub-systems and identify the dependencies. Then convert the sub-system into components. Usually, there is a need for the system to operate during the conversion. This approach narrows the exposure to a sub-system, which can be isolated and managed. Also, you may have to maintain the conventional infrastructure, such as massages and queues, until a sub-system is completely converted into components. This problem becomes particularly difficult when resources are not available for the whole sub-system.

The next step is to create a component specification, which includes the interfaces and member functions. The objective is not only to contain logically related interfaces, but also perform load balancing. There are two types of load balancing, technical load balancing and project management load balancing. Technical load balancing requires stack space consideration and real time considerations. Project management load balancing requires matching the organizational structure to component-based software engineering. Would an individual own a component or would a team of engineers own a component?

The granularity of the components will depend on the size of the software project and the number of people working on the project. It is not unreasonable to have seven to ten interfaces per component with the same number of member functions per interface. Again, the size of the project and the size of the organization will dictate the granularity of the components. It would me most efficient to match individuals and teams to components. An organization matched to create components will be most efficient.

Once the component specification is complete, the developers have complete control of the development method or approach, as long as the interfaces are adhered. During development, the metrics used to track progress should be based on component-based software engineering. Traditional metrics, such as Lines of Code (LOC), is no longer valid or useful. Metrics based on interfaces, member functions and their complexity should be used. Since few guidelines exist today, the initial metrics for component-based software engineering would have to be devised. Further, each problem domain may require unique metrics. Naturally, the number of components, the number of interfaces, the number of member functions and their complexity will form the basis for metrics.

Unit testing in component-based software engineering can be accomplished by creating a sister test component for each functional component, which exercise the interface and its member functions. A test harness could used to automate the test process. The creation of sister test components can be automated by customizing the IDL compiler. Component integration testing can be accomplished by isolating logically related components and identifying their interface. Component-based sub-systems can be tested similarly.

Guidelines, which require the creation of sister test components before functional components can be checked-in, would formalize the unit test strategy. Tools to automate the creation and execution of sister test components would enable software engineers to follow this strategy.

Maintenance of component-based software systems can be revolutionized. In conventional software systems, coupling enslaves engineers to the implementation. Their frustration is further exacerbated when simple changes in one area of the code causes dramatics effect in other areas of the code. Maintenance becomes a curse and engineers are tethered to the implementation. Creativity is stifled. Much of our engineering talent is sacrificed to maintenance. In component-based software systems, engineers can break their chains. They can make changes to the component with confidence and, if necessary, change the implementation altogether, as long as they are faithful to the interfaces.