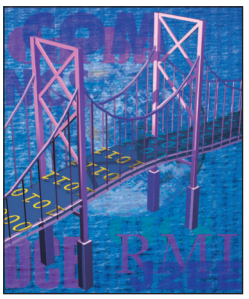

Old Measures for New Services

Steve Vinoski • *IONA Technologies*

Like fashion and music, computing styles tend to come and go and then come back again. I often think of this phenomenon as being like a pendulum, continuously swinging from one side to the other. For example, popular user interface (UI) styles vary every few years. "Rich client" UIs are currently all the rage, but a few years ago, the "thin client" approach was favored, and a few years from now you can be sure it will be again. Mainframes ruled the roost in the 1970s, but their popularity dwindled through the 1980s as minicomputers, workstations, and personal computers appeared on the scene. I recall lunch-table discussions around 1990 in which the popular stance was that the mainframe was dead. Contrary to those opinions, mainframes quietly rebounded throughout the '90s, and today they still sell well.

Older computing styles can reappear for different reasons. Sometimes advances in hardware, programming languages, operating systems, or middleware breath new life into techniques that didn't work so well in the past. In other cases, nearly forgotten methods return to popularity because the "not invented here" syndrome forces a backlash against current approaches, or because a lack of awareness that such approaches had been tried in the past makes them seem fresh and new. Sometimes, new approaches simply don't work out, forcing a return to approaches that previously worked. Of course, there are even cases where once-popular tactics make comebacks because they're fundamentally good ideas.

Back to Services

Service-oriented architecture (SOA) is somewhat overhyped at the moment, making it seem new when it really isn't. After all, popular RPC toolkits from the late 1980s to early 1990s, such as Sun's Open Network Computing (ONC) and the Open Software Foundation's Distributed Computing Environment (DCE), were oriented around services, as were later distributed object systems such as COM, Corba, and Enterprise Java Beans (EJB). It's enough to make you wonder if there are any new techniques for designing good services, or whether we've already encountered them all in previous computing-style waves.

Preceding the object-oriented programming wave was the structured programming approach.[1,2] It began in the early 1970s with the hope of turning the seemingly haphazard craft of programming — with its spaghetti code and mischievous `GOTO`s — into true engineering, with distinct phases and repeatable processes. The end goal was, of course, to produce higher-quality software. Structured programming led first to structured design and then to structured analysis, with levels of abstraction continually rising along the way. Structured techniques promoted various types of diagramming to allow for true software engineering designs that were free of the "encumbrances" of implementation details. Programmers first broke problems down using stepwise refinement; once all the steps were implemented, they could simply be assembled to form the solution to the original problem. Although far from perfect, structured techniques taught us a great deal, both in terms of what worked and what to avoid.

One of the most useful things that structured techniques gave us was an approach for measuring software quality based on the fundamental concepts of *coupling* and *cohesion*. Coupling is a measure of interdependencies between modules, which should be minimized; cohesion, a quality to be maximized, focuses on the relationships between the activities performed by each module. You might not hear much about these old measures anymore, but they're still highly applicable to today's service-oriented systems.

 Published by the IEEE Computer Society 1089-7801/05/$20.00 © 2005 IEEE

Coupling

Given that coupling is something to be minimized, quality systems tend to exhibit "loose" rather than "tight" coupling. Unfortunately, these measures are too black-and-white. According to Meilir Page-Jones's 1988 *Practical Guide to Structured Systems Design*,[1] multiple forms of coupling exist along the scale from loose to tight, or clear to concealed.

- *Data coupling* occurs when modules pass data between each other, such as when a variable is passed as a parameter to a function call. There's nothing too unusual about this form of coupling; although you can go overboard with it (passing many parameters to a function, for example), data coupling is generally preferable to the other forms of coupling.
- *Stamp coupling* is like data coupling but with composite parameters. The difference is that the composite structure's fields are passed somewhat invisibly, making their relationship to the called module less clear than with those passed directly. Passing a whole structure allows the called module to manipulate all fields contained therein, which can be undesirable if the module really only needs access to one or two fields.
- *Control coupling* occurs when one module knows something about and passes information that's intended to control the logic of another. While it's not unusual or undesirable for a module to use its parameters to make decisions about its actions, it's less than ideal when this leads to decisions' being split across modules. Not only does such an arrangement make it harder to change the decisions, it also implies reduced module cohesion as well.

These forms, which Page-Jones collectively refers to as *normal* coupling, are generally okay, except for the specific caveats I mentioned. Other forms, however, are to be avoided:

- *Common coupling* occurs when two modules share a common data area, such as global variables. This needs no explanation, as we all know the evils of globals. Common coupling might seem immaterial when considered in the context of today's distributed services, but it becomes very relevant when you consider the resemblance to the coupling that occurs when services share directories, registries, or databases. They're not necessarily identical forms of coupling, because directories, registries, and databases are themselves services that aren't completely wide open like a global variable. That said, it's sometimes tantalizingly easy to create systems in which services share data at will through common registry entries or database tables, which isn't much better than using global variables.
- *Content coupling* describes the situation in which one module directly refers to another's innards. Like global variables, this form was originally applied only to functions sharing address spaces. However, its more modern form is generally known as *implementation coupling*, in which one module depends on the implementation details of another. This is arguably the worst possible degree of coupling, given that it potentially includes all the other forms mentioned, and it means that changes to one module or service will almost certainly require modifications to the other.

Another form of coupling that's not mentioned in structured programming texts might be called *interface coupling*. Highly relevant to distributed objects and service-oriented systems, this form refers to the degree to which one module depends on another's interface. The main reason this never came up with structured programming techniques is that they equated "module" with "function," where a function has only a single interface, which means depending on it is a given if it's being invoked. However, interface coupling becomes a much larger concern when the term "module" is used in its modern form, in which it's equivalent to an object or, bigger still, a package or namespace. Objects and packages generally offer multiple methods, and it's not unusual for code using an object or package to depend on many or all of its methods. Interface coupling is synonymous with the term "surface area," which is an informal measure of how much of a given interface the code using it depends on.

Although far from perfect, structured techniques taught us a great deal, both in terms of what worked and what to avoid.

Another facet of interface coupling relates to the number of interfaces that a given module or application depends on, and the distinctiveness or uniqueness of each interface. For example, an application built entirely around one or two polymorphic interfaces exhibits low interface coupling, regardless of how many actual objects or services it interacts with through them. An application that depends on a wide variety of unique interfaces, on the other hand, is highly coupled.

Cohesion

Just as there are different forms of coupling, there are different forms of

cohesion. Page-Jones describes seven different levels. Three of the forms avoid creating forced or unnatural relationships between a module's internal tasks or the data it uses:

- *Functional cohesion* occurs when a module does only one thing. This is the ultimate in module cohesiveness. Functionally cohesive modules that also display low coupling are typically highly reusable because they are, by definition, self-contained and largely independent of any surrounding code.
- *Sequential cohesion* occurs when a module carries out several tasks, and the input of one task feeds into another, perhaps modifying the data as it passes through. A sequentially cohesive module or service is often a wrapper around other modules or services, chaining them together to perform a larger function. This, in itself, isn't necessarily undesirable because good wrappers can hide details, thereby reducing overall coupling within applications.
- *Communicational cohesion* is when a module carries out multiple operations based on the same input or output data. Such cohesion often results from the desire to operate only once on a complex set of data, calculating in one pass everything that the rest of the application might possibly want to retrieve or derive from it. Unlike sequential cohesion, task order is unimportant in communicational cohesion.

In contrast to the "good" forms of cohesion, Page-Jones identifies four that are frowned upon:

- *Procedural cohesion* is similar to sequential cohesion except that the data feeding each of the tasks is different. Such cohesion often results from artificially grouping activities of an application together into catch-all functions in a misguided attempt to reduce coupling.
- *Temporal cohesion* occurs when a module's tasks are related only by the time they're carried out. Such modules cause maintenance problems if one of the tasks needs to be performed at a different time.
- *Logical cohesion* is a condition in which a module's activities are grouped together because they appear to be able to share common implementations. This results in a strange or awkward interface for the module, thereby punishing all of its users, just to ease the implementer's job.
- *Coincidental cohesion* represents the bottom of the cohesion barrel, in which a module's tasks are related only by the fact that they reside together in that module.

Just as with coupling, cohesion matters when applied to distributed objects and services.

Just as with coupling, cohesion still matters when applied to distributed objects and services. For example, if you grouped a bunch of methods in an object only because they had similar implementations, you would be guilty of creating a logically cohesive object.

As you (re)acquaint yourself with the various forms of coupling and cohesion, their applicability to today's service-oriented systems becomes clear. For example, consider the seemingly unending debate about all the object-like Web services specifications, typically referred to as "WS-*," versus the Representational State Transfer (REST) approach on which the World Wide Web is built. If we analyze these approaches from a coupling point of view, we find that although both exhibit similar forms of data coupling, the degree of interface coupling exhibited by systems based on WS-* is much higher than for REST systems. This is because REST's interfaces are uniform and fixed, whereas WS-* interfaces are ad hoc and variable. As another example, creating understandable and maintainable Web services orchestrations requires us to consider the cohesion of both the services being orchestrated and the resulting orchestrations themselves. It would seem much too easy, for example, to create orchestrations with procedural cohesion, rather than applying the extra analysis and design effort to create sequentially or even functionally cohesive service groupings.

Given that transitions to "new" computing styles are often accompanied by explicit disapproval of the outgoing style, it's no surprise that today's focus on SOA has created a bit of a backlash against distributed objects. What's unfortunate is that many of the measures of quality for distributed object systems apply equally well to distributed services and SOA, so it's a shame that some feel compelled to ignore them just to be trendy. But perhaps it doesn't matter, because we can just go back to the days before objects, dig up measures like coupling and cohesion, and apply them all over again — for the first time, of course.

References

1. M. Page-Jones, *The Practical Guide to Structured Systems Design*, 2nd ed., Prentice-Hall, 1988.
2. J. Martin and C. McClure, *Structured Techniques for Computing*, Prentice-Hall, 1985.

Steve Vinoski is chief engineer for IONA Technologies. He's been involved in middleware for more than 17 years. Vinoski has helped develop middleware standards for the Object Management Group (OMG) and the W3C. Contact him at vinoski@ieee.org.