Promoting Component Architectures in a Dysfunctional Organization

by Raj Kesarapalli
Product Manager
Rational Software

When I first began my career as a software developer, I didn't quite understand what component architecture was all about. But after spending a few years doing software development, I now have a deep appreciation for it. As it turns out, there is only one right way to develop software, and that is by using component architectures. Unfortunately, it is still far from a universal practice. When I talk to my friends in Silicon Valley about component-based development practices, they all seem to understand it. But when I probe further, they all complain about the poor coding practices at their respective companies. They are all too familiar with the terms "band-aid" and "spaghetti code."

As developers, most of us at some point have worked late hours debugging someone else's code, because there was no way to debug our own code in isolation. In the age of component architecture, this should be a thing of the past, but shorter release cycles and deadline pressures cause developers to take shortcuts that defeat the promise of component architecture for a development team. In many cases, the initial designs for a team-coding environment are based on component architectures: All the major functionality is well componentized and meant to be tested in isolation. Over time, however, most of those systems initially based on components stray from the original designs, resulting in a monolithic piece of code that is hard to debug, test, and reuse. When this happens, the result is frustrated teams and delayed projects.

Why does a project team stray from an initial component-based design? If you are a developer frustrated about your teammates not adopting proper coding standards, how can you address that without sounding like a know-
it-all, or worse, insulting your colleagues? And if you are not a hands-on
development manager, how can you be assured that your team is
following the best practices regarding component-based architecture?

During the ’90s, I had an interesting experience trying to promote
component architectures and code reuse. In this article, I’ll explain the
hurdles I ran into and the approach I took to promote component
architectures and code reuse. I hope you’ll find it useful.

The Diverse Team Environment

Today, development teams are made up of developers from different
backgrounds with different experiences and motivations. They do not all
think alike. At first glance, everyone may seem to understand component
architectures and code reuse very well, but everyone will interpret these
things differently, and without proper caution, the resulting code will be
hard to debug, test, and reuse.

I learned this the hard way when I spent a year at a dotcom developing a
Windows-based application that lets users run applications over the
Internet without having to install them on a desktop. When I first started
there, the application was well designed; it was organized into many
modules, each representing a core piece of functionality with well-
designed APIs that other modules could call. (I call them modules as
opposed to components because they are implemented as libraries with
exposed APIs, not COM components. You can think of these modules as
logical components.) Everyone on the team was assigned a module, and
we held design reviews to discuss proposals and agree on the APIs.
According to our agreements, team members would use only these APIs to
call into a given module. Our agreements were based on trust, and this
approach worked fine -- initially.

Over time, requirements changed, so we needed to add new functionality.
As usual, time was short and pressure was great, so instead of carefully re-
designing the application and re-designing the modules, developers took
shortcuts. In the process of adding the new functionality, developers
created dependencies between different modules by accessing data in
other modules directly. They’d change some of the private methods to
public methods to borrow functionality instead of moving that functionality
to a shared module (as good code reuse practice dictates). In other cases,
they’d borrow functionality by duplicating code in multiple locations, thus
creating multiple instances of the same bugs. Over time, some modules
that should have been re-designed and broken down into multiple modules
grew monolithically huge instead.

Because of the dependencies introduced between modules, unit testing
and unit test development became too cumbersome and time consuming.
Eventually, the project team did away with unit testing, which meant you
had to debug the entire application. In my case, this was extremely
painful: Debugging the entire application meant rebooting the machine
every few minutes. Had I been able to test my module in isolation, I
wouldn’t have had to do all this rebooting, which resulted in long,
unproductive debugging sessions. And when new developers moved on to
the project, they had a very rough time coming up to speed and invariably introduced many new bugs in the process.

**Keeping Teams Aligned with Component Architectures**

At the next opportunity to add functionality, I created a new module (a library). This time, I didn't want to run into the same problem, so I took a different approach when my module was ready to be added to the application I was working on. By then, we had two other applications under development, so in addition to adding my new module to the project build that I was working on, I also added it to the other two application builds. The other teams knew they would eventually need that functionality, and since I did all the upfront work to make sure that all three applications built fine with the new module, they didn't have an issue with the early addition.

I added my new module right away (as soon as I created it) as opposed to waiting until my colleagues needed it for several reasons. First, bear in mind that my module was now part of the builds for three different applications. Each time a developer took a shortcut and added dependencies in my module to other modules, the immediate product the developer was working on built fine, but the other two failed. This forced developers to make changes to my module the right way -- coding three different shortcuts to fix the problem in the three builds is harder than coding once the right way. It prevented developers from taking shortcuts and making mistakes, and it helped my module remain componentized, so maintaining it was a breeze. Since it stayed componentized, the module was always ready for re-use, and new projects used it right away. One year later, the module was being re-used in seven different projects. This would have been impossible if I hadn't created that reuse situation up front.

**Component Architectures from a Manager's Point of View**

Component architectures promote code re-use, and, conversely, once a development team commits to the concept of code re-use, it becomes relatively easy to adopt the principles of component architecture. The challenge for most development teams lies in continuing to follow the principles of component architectures over a product's life cycle.

If you are a manager wanting to make sure that your team is developing code the right way, here is something you should know: Developers have a lot to focus on, and your average developer doesn't think of component architectures and/or code reuse unless asked or perhaps forced to do so. The average developer is more likely focused on getting the work done as quickly as possible before the upcoming deadlines.

As a manager, you should invest some time and effort in creating an environment in which it's hard for developers to make mistakes. The programming languages and IDEs we use today don't enforce the
principles of component architectures. And even in cases where these environments do support component architectures, there is additional work that developers need to do -- for example, in some popular IDEs, many developers feel that the frameworks supporting component architectures are restrictive and time-consuming to work with, and this is enough to prevent them from building proper components. Fortunately, as programming languages and IDEs become more sophisticated, the additional work that developers must do manually today will be automated in the future. But until then, it is up to you as a manager to make sure your team uses proper techniques for code reuse.

Consider also the scope of a given component; for example, what is a well-designed component? How much code should a given component contain? Make sure all API changes are reviewed and designed. It will make sense to add some changes to existing modules, and some changes will require the creation of new modules. The trick is to promote reuse early in the design phase, because designs that factor in reuse result in good components. This is good management practice, and it will lead to good coding practice as your team adopts the principles of component architectures and code reuse.

**Try Unit Testing**

If you don't have a reuse situation as I did, try unit testing to help you keep your modules componentized. If the unit test breaks at any given point, it is likely that someone coded dependencies into the module that don't belong there. The key is to have the developer create unit tests (at least one) before the module is made available to the rest of the team.

Developers often skip unit tests, complaining that unit test development is difficult and a waste of time. You should pay careful attention to such complaints. If a module is based on component architecture, unit test development should be trivial. These complaints may be a tip-off that the damage is already done, in which case creating a system for unit testing will represent a huge investment of time and human resources. If this is, in fact, your situation, then you should at least identify the few core functional pieces (which are usually the candidates for reuse) and componentize them one by one over time. Once you componentize them, you can add unit tests to each of these modules to keep them componentized. This will greatly help you localize bugs within modules, which means you can debug your own module in isolation, as opposed to debugging the entire application. And because developers will only change code they are familiar with, they will be less likely to introduce new defects. This will lower the defect count and reduce maintenance costs.

**Parting Thoughts**

As I mentioned earlier, component architectures promote code reuse and make unit testing trivial. Conversely, unit tests ensure your code stays componentized. In combination, these practices will have a positive long-term effect on your code.

Note: If you have other strategies for introducing best practices that you'd
like to share, I'd be interested in hearing about them. You can reach me at Raj.Kesarapalli@rational.com

Notes

1 Many managers take unit tests for granted, and all but ignore the results. By implementing unit tests, you are effectively creating a reuse situation similar to what I did by adding my module to two other project builds. It's even simpler to use unit tests to achieve the same goal.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!