Modules: Keeping Things Loose

Posted on **August 1, 2012**

A recurrent theme in this blog has been the concept of modular programming. The systematic deconstruction of large complex applications into more manageable chunks know as modules. Each module operates by being an abstraction of a key concept in the application. These in turn interact with each other to create the desired total application. It is the interaction of these modules that is the topic of this discussion.

Whenever modules interact, there is a sharing of information that must happen. For example, if module “A” calls a function in module “B”, then module “B” must share with module “A” the signature of the function in question. In addition, the purpose of the function must be clear so that it may be called appropriately. Another example would be module “A” accessing a variable in module”B”. This causes a great deal of shared information, some of which are, the signature of the variable, the meaning of the data it contains, how mode “B” reads the variable, how module “B” updates the variable, how module “A” and all other modules read and update the variable, what actions are based on the variable, what events affect the variable, and list list goes, seemingly without end.

In the first case, the function call, where little information is shared, is referred to as loosely bound or coupled. The analogy being that of an arms length business relationship. The second case, where a great deal on information is shared is called tightly bound or coupled. The human analogy being that of a romantic couple.

Now, before the orthodox view is examined, it must be pointed out, that tightly coupled modules do have their good points. When modules work very closely together, a great deal of “protocol” overhead may be omitted and high levels of performance can be achieved. This is especially so in highly resource constrained systems. The down side is clear in the above statement though. By working as one, the two modules have in fact become one big module. Taken to the limit, the application becomes one huge mass of code where changes to one area have undesirable side effects in seemingly unrelated areas. Again, highly resource constrained systems are often quite small, and thus have smaller/simpler applications. In these cases this complexity may be manageable.

The predominant view is that loose coupling is better. Most systems are NOT hyper-constrained and applications are large and complex. By keeping the inner-working of modules hidden and private, many benefits are observed.

- Clarity; when modules interact along clear, well defined lines lines, the code becomes easier to understand, write, and debug. Further, loose coupling generally results in simpler module interfaces are easier (and thus more likely) to document clearly. Tightly coupled modules require an extensive documentation effort that is often never done.
- Maintainability; by limiting access, the odds of a change in one module have an undesired side-effect on an other are great reduced.
- Reuse: When modules are loosely bound, it is possible to reuse modules in different applications without having to “fix” all of the “back-door” interactions.
- Abstraction; having a limited, published interface means that the module is able to provide an abstract view of the services provided, shielding the consumer modules from the details of operations.
- Team Programming; when modules have well defined interactions, it becomes easier to divide the work among a number of coders. Since each need only concern themselves on the published interfaces they use, it is no longer necessary to memorize the entire application and programmers are less likely to “step” on each others work.

The downside of loosely coupled systems is that some performance is lost “going through channels” rather than taking short-cuts. In my career, I have sometimes seen this argument put forth as a reason to abandon the discipline of loosely coupled modules with well defined interfaces. These factors will all have to be considered in the design of each interface, still, the overwhelming advantages of loose coupling should make it the default choice that is only set aside when circumstances leave no choice.

This article has dealt with module interaction in general. An important special case is where one module (the source) needs to transmit data or an event to another module (the sink). That is the topic of the upcoming post “Modules: Push or Pull”.

After a long hiatus, this web site is back! I want to take this time to apologize to my readers for the long pause in postings. A lot of water has gone under the bridge (and some even over it) in my life, leaving no time for my duties here. Hopefully things will be better now. As always your comments, thoughts and suggestions are most welcomed.

Peter Camilleri (aka Squidly Jones)

LIKE THIS:

Like

Be the first to like this.

This entry was posted in **Embedded Development** by **Peter Camilleri**. Bookmark the **permalink [http://teuthida-technologies.com/?p=1034]** .