Issues to Consider Before Acquiring COTS

Dr. David A. Cook
The AEgis Technologies Group, Inc.

In today's software and acquisition environment, the decision to use commercial off-the-shelf (COTS) and government off-the-shelf (GOTS) is not only necessary from a schedule and cost point of view, but often is mandated. However, there are many factors that influence the decision to use and choose COTS/GOTS software. Readers who have a background in computer science or who have taken some formal software engineering code development classes are typically familiar with the effects, but many engineers who acquire COTS do not have this background. This article discusses some basic but often-neglected factors affecting COTS selection and use.

Back in the early days of computing, COTS use was commonplace. However, instead of integrating with other programs, the COTS packages ran independently. It was commonplace to lease or buy accounting or other similar software from your computer vendor. Locally written software seldom, if ever, interfaced with COTS. Programs tended to be relatively small, and interaction between programs consisted primarily of the output of one program being fed in as input to other programs.

As computing power increased, and computer use proliferated, more and more programs interacted with other programs, and all interacting programs ran concurrently. As this occurred, software manufacturers filled the need for common programs by devising programs that were not made to run independently, but instead provided partial solutions to the overall problem. The intent was that you could purchase a product that would meet one or more of your requirements and simply plug in this product.

COTS was, at one time, envisioned as a massive time and money saver. You would go shopping at a software warehouse and select COTS systems as needed. It would seamlessly interact with your organic, homegrown program and also with any and all other COTS products that you selected. In a sense, COTS would operate like the ubiquitous Universal Serial Bus (USB) *plug and play* devices that have become so common.

Unfortunately, this promise of *plug and play* software was not as effortless as *plug and play* was for hardware. In hardware, there are very specific standards that specify how the interface to the computer must occur. This allows a hardware creator to design a product that is specific in its purpose (for example, a 256 Meg flash drive) but general in its interface (so that when inserted, drivers automatically load with any disk insertion or user interaction). In hardware, there are general device drivers that have extremely specific interface standards. Plug in a USB drive from almost any manufacturer and the device will work without any specific *tailoring* of the device to the hardware.

COTS, on the other hand, typically do not have specific standards. COTS requires the user to adapt his/her environment to use the COTS. Whereas the *plug-and-play* hardware has a known and generalized standard that all users are willing to adapt to, COTS requires each user to adapt their individual software to interface with the COTS. Each user typically has specific and unique needs, yet they want to interact with general-purpose COTS. This usually causes problems in that the COTS does not meet all of the user needs, and in fact might contain functionality that the user does not want. Explaining how COTS interacts with other applications can be better understood by examining two relatively common software engineering concepts: coupling and cohesion. Coupling and cohesion, which are relatively basic terms in computer science, can be applied to COTS to help determine COTS quality and also to determine how easy it will be to integrate COTS with locally developed code.

Coupling

Coupling refers to how programs interact. There is a range of how multiple modules interact (see Table 1 for the different types of coupling). In the best

Table 1: *Types of Coupling, Ranked From Best to Worst*

Message Coupling (low coupling, and the best type)	This is the loosest type of coupling. Modules are not dependent on each other; instead, they both use a public interface to pass messages between them, such as an object-oriented message.
Data Coupling	Data coupling is when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data that are shared (e.g. passing an integer to a function that computes a square root).
Stamp Coupling (data-structured coupling)	Stamp coupling is when modules share a composite data structure and use only a part of it, possibly a different part (e.g. passing a whole record to a function which only needs one field of it). This may lead to changing the way a module reads a record because a field that the module does not need has been modified.
Control Coupling	Control coupling is one module controlling the logic of another by passing it information on what to do (e.g. passing a what-to-do flag).
External Coupling	External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface (e.g. you have no control over the interface).
Common Coupling	Common coupling is when two modules share the same global data (e.g. a global variable). Changing the shared resource implies changing all the modules using it.
Content Coupling (highest coupling, and the worst type)	Content coupling is when one module modifies or relies on the internal workings of another module (e.g. accessing local data of another module).

Source: <http://en.wikipedia.org/wiki/Coupling_(computer_science)>

scenario, modules each fulfill a specific purpose and no direct interaction is required. This lack of coupling is perfect in that the actions of one program or module have little or no effect on other programs. However, this type of scenario is reminiscent of olden days when programs ran sequentially, rather than parallel. Today, COTS almost always requires interactions with other concurrently running programs.

Prior to C++, the best type of coupling was referred to as *data coupling*. In data coupling, all interaction is defined by parameter calls. This type of coupling is simple and the least likely to cause a ripple effect – that is, when a change to the logic of one module causes undesirable effects to other modules. The advent of commonly used object-oriented languages has led to a new (lower, and therefore better) level of coupling, referred to as *message coupling*[1]. In a well-designed system, low coupling gives COTS the ability of *plug and play* in terms of a standard interface. With only *message coupling* or *data coupling*, it should be relatively easy to remove one COTS module and replace it with a similar COTS module that has the same interface.

Back in the 1970s, the first car I owned – a 1973 Chevrolet Impala – had an AM/FM radio, but I wanted to replace it with an AM/FM/Cassette radio. After disassembling the dashboard (no small feat), I removed the radio to find out that it had six black wires, one white wire, and one green wire. The new radio had seven black wires and a white/green twisted pair. I had no idea how to cleanly replace one radio with the other. A more recent car, bought a few years ago, had a standardized harness plug. When I asked to replace the standard AM/FM/Cassette/CD with a six-CD radio, it took about 10 minutes – basically just unplug the old radio and plug in the new. This is the kind of interfacing you want with your software: easy to uninstall the old, easy to install the new. When your current COTS supplier goes out of business or no longer supports your version, it should be easy to replace.

It is not important, of course, to determine exactly what the level of coupling is – it is more important to keep it low[2]. Coupling is directly related to information hiding, along with data abstraction and data design. The moral of coupling is to *really* design systems that interact (or might interact) with COTS. Use interface control documents. Have an architectural design document. Review data requirements with the user, have a data dictionary and data design document, and make the COTS interfaces as simple and straightforward as possible. Coupling should be kept as *low* as possible.

Table 2: *Types of Cohesion, Ranked From Best to Worst*

Functional Cohesion (high, and the best type of cohesion)	Functional cohesion is when parts of a module are grouped because they all contribute to a single, well-defined task of the module (e.g. calculating the sine of an angle).
Sequential Cohesion	Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part like an assembly line (e.g. a function which reads data from a file and processes the data).
Communicational Cohesion	Communicational cohesion is when parts of a module are grouped because they operate on the same data (e.g. a module which operates on one record of information).
Procedural Cohesion	Procedural cohesion is when parts of a module are grouped because they always follow a certain sequence of execution (e.g. a function which checks file permissions and then opens the file).
Temporal Cohesion	Temporal cohesion is when parts of a module are grouped by when they are processed – the parts are processed at a particular time in program execution (e.g. a function which is called after catching an exception which closes open files, creates an error log, and notifies the user).
Logical Cohesion	Logical cohesion is when parts of a module are grouped because they logically are categorized to do the same thing, even if they are different by nature (e.g. grouping all input/output handling routines).
Coincidental Cohesion (low and the worst type of cohesion)	Coincidental cohesion is when parts of a module are grouped arbitrarily (at random); the parts have no significant relationship (e.g. a file of frequently used functions).

Source: <http://en.wikipedia.org/wiki/Cohesion_(computer_science)>

Cohesion

Cohesion refers to the measure of how COTS performs a single task. It is a measure of the *stickiness* of the module. A good module contains only resources that accomplish single or similar tasks. The parts of the module are all closely interrelated and therefore stick together. Table 2 explains the types of cohesion.

Many studies have shown that coincidental cohesion (such as one routine that would *calculate tax rate and compute Celsius to Fahrenheit*) and logical cohesion (such as *open all input files*) are extremely inferior to other types of cohesion[3]. Combining many functions into one module makes cohesion low and contributes to high error rates and increased debugging, testing, and integrating time. In addition, relatively small maintenance fixes tend to have ripple effects that require testing and debugging of many routines that are not logically related. Good developers know that an application that has many relatively simple modules is easier to develop and test. In fact, this is one of the basic concepts of object-oriented programming.

For most computer users, the programs we use have a single purpose. Most computer users have learned that it makes sense to have a single word processor, a single spreadsheet program, etc. While they might all come from the same software developer and be designed to interact together, they are still separate programs. Back in the 1980s, there was a push to deliver *all-in-one* software that combined lots of functionality. It was clumsy to use, performed poorly, and was widely hated by most users. Instead, single-purpose applications (which had high cohesion) that had well-defined *cut and paste* and hyperlink interfaces (low coupling) tend to be easier to use, to be more robust, and to include more functionality. COTS works the same way. A single-purpose program typically works better and is almost always easier to integrate.

Cohesion is directly related to modularity (whereas coupling was directly related to information hiding). In a well-modularized program, routines are reasonably small and perform one action. In a poorly modularized program, routines grow very large – sometimes thousands of lines. Good software engineers know that it is easier to write, test, and

then integrate lots of small routines rather than try to write and debug monolithic modules of several thousand lines.

Whereas it is important to keep coupling low, cohesion should be kept high. To design for high cohesion, architectural and module design documents are needed. Good requirements are a must, as are traceability matrixes. A traceability matrix allows you to map the requirements directly to the design and module and encourages you to have small modules that meet a small number of requirements.

Again, there are metrics to measure cohesion, just as there are metrics to measure coupling. However, it is not as important to measure coupling or cohesion; it is more important to try and keep coupling low and cohesion high. With COTS, you want to insert off-the-shelf functionality that provides both single and well-defined functionality.

COTS Testing and Requirements

It is obvious in hindsight, but how do you *know* that the COTS products that you are buying will meet your needs? Certainly you cannot believe the marketing hype, and because COTS products are developed for many users, your application might have unique COTS needs that have never been tested for or used in other applications. The solution is to have a well thought-out test plan that makes sure that your unique needs are tested. This test plan needs to be developed *before* you acquire COTS. And, to develop a good test plan, you need good requirements. The bottom line is that if you acquire COTS before you have high-quality requirements that have been validated by the user, you are likely to end up with COTS that do not meet all of your needs.

Of course, maybe there are no COTS products that meet all of your needs, anyway. This is a common occurrence. There are several solutions: Either modify the COTS or develop so-called *glue* code that holds the COTS together.

I highly recommend against attempting to modify COTS. Such attempts typically lead to increased cost and lengthy testing[4]. Instead, consider either devising a manual work-around or implement a limited amount of *glue* code. However, keep in mind that if the *glue* code is complex and hard to maintain, it might overcome the cost and time savings of using COTS.

Version, Speed, and Licenses

Other issues affect the overall cost of a COTS system. Systems with a lot of COTS components have a problem with versioning, both with the versioning of the COTS and with the underlying operating system. When COTS is updated by the vendor, it requires careful regression testing to make sure that the newer versions of COTS continue to meet the exact *form, fit,* and *function* of the previous version. In addition, care must be taken that newer versions, as they are released, do not add unwanted functionality (and vulnerabilities).

A second issue is that in addition to successive versions of newer COTS, each COTS package might want specific versions or patches to the operating system. Each might want a separate version of the Java Machine installed. As updates to both the operating system and system libraries are installed (in some cases automatically), how do you know that the COTS will continue to work? Again, a test team must be continually testing and checking to make sure that updates to the operating system do not cause unintended side effects. It helps if the COTS has a scheduled update cycle and if pre-release versions of the upcoming COTS can be tested early.

"I highly recommend against attempting to modify COTS. Such attempts typically lead to increased cost and lengthy testing."

A third issue with versioning becomes important when you have multiple COTS running. As the matrix of interrelated COTS packages are updated, interfaces from one COTS package might interfere with other COTS interfaces. System and hardware requirements might conflict, or be vastly magnified with multiple COTS packages installed.

To address all of these issues, a test plan (and a dedicated test team) is critical. Research into the interconnection between various COTS packages is required on a continual basis. In the past, the principle was typically *get it running and lock down the entire configuration* and baseline the system (and prevent continual COTS and operating system updates). However, because of security threats, operating systems of today require continual updating to meet increasing security threats. It is no longer practical to *lock down* an environment.

Other issues with COTS involve the performance of the integrated system. In any system, it is important to test (or at least prepare for) *worst case* conditions. Unfortunately, as requirements change during development, worst case conditions sometimes change, also. When you acquire COTS for a specific need and the need changes during development, performance problems can occur. The performance/speed issues can also involve hardware. Remember that you need to develop and test the entire system for not only the average user, but also the worst-case user. Finding out that the users have inadequate hardware after development and release of the software is bad. Also, remember that hardware preparation also includes network support. It would be catastrophic to deliver a finished system only to find out that there is not enough available bandwidth necessary to support the new COTS database. Again, this is something that can be avoided by having good requirements and by having a test team testing against the requirements.

A final issue is licensing costs. For some COTS, a separate license must be acquired for each user. Do you know how many end-users will need licenses? Is there an acquisition mechanism in place to buy and distribute the licenses when you release the COTS? Proper preparation for release requires that you make plans for licenses and license distribution.

These are just a few issues that affect the overall cost of COTS. In this issue of CROSSTALK, the article *Added Sources of Costs in Maintaining COTS-Intensive Systems* by Clark and Clark explores these additional costs in greater detail.

Conclusion

COTS acquisition and integration are complex topics – this article simply touched on some of the basic issues to consider. Coupling and cohesion are relatively basic computer science concepts, but an understanding of how they relate

to good software engineering can make the job of acquiring COTS easier. From a larger perspective, basic COTS questions can be answered by basic *good* software engineering: Gather good requirements, validate the requirements, perform good design, and have a test plan. COTS probably will not solve 100 percent of your requirements, so it is important to know what you are willing to settle for. Also, understand that COTS might save you time and money, but trying to integrate it incorrectly can cost you lots of money and will not necessarily reduce the schedule time. Trying to integrate multiple COTS applications will probably magnify your potential problems; so learn from the mistakes and successes of others whenever possible. Be wary of marketing hype, and remember that finding another user with experience with your potential COTS product is one of the best resources of all.◆

Notes

1. For a discussion of program language evolution and programming language interoperability, refer to "Evolutionary Trends of Programming Languages," by Thomas Schorsch and David Cook in CrossTalk Feb. 2003 at <www.stsc.hill.af.mil/crosstalk/2003/02/schorsch.html>.
2. See "Software Engineering, A Practitioners Approach" by Roger Pressman for a good description of coupling, cohesion, information hiding, abstraction, and modularity. You can also get pointers to online sources of information by checking out *coupling* and *cohesion* on Wikipedia.
3. For example, see E. Yourdon, and L. Constantine. "Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design." Prentice-Hall, Yourdon Press.
4. For an example, see "Business Process Reengineering/Software Modifications" OIG/97E-10 – "Evaluation of Best Practices for Developing and Implementing Integrated Financial Management System From the U.S." by the Nuclear Regulatory Commission, available at <www.nrc.gov/reading-rm/doc-collections/insp-gen/1997/97e-10.html>.

About the Author

David A. Cook, Ph.D., is a senior research scientist at AEgis Technologies, working as a verification, validation, and accreditation agent in the modeling and simulations area. He has more than 30 years experience in software development and management. Cook was associate professor and department research director at the U.S. Air Force Academy and former deputy department head of the software professional development program at Air Force Institute of Technology. He was a Software Technology Support Center consultant for six years. Cook has a doctorate in computer science from Texas A&M University.

The AEgis Technologies Group, Inc.
6565 Americas Parkway NE
Albuquerque, NM 87110
Phone: (505) 881-1003
Fax: (505) 881-5003
E-mail: dcook@aegistg.com