Added Sources of Costs in Maintaining COTS-Intensive Systems

Dr. Betsy Clark and Dr. Brad Clark
Software Metrics Inc.

Ten years ago, work was begun at the Center for Systems and Software Engineering at the University of Southern California to develop a cost model for commercial off-the-shelf (COTS)-based software systems[1]*. A series of interviews were conducted to collect data to calibrate this model*[2]*. A total of 25 project managers were interviewed; for eight of these projects, data was collected during the original system development and maintenance phases. A common sentiment heard from the people maintaining these systems was that they turned out to be more expensive to maintain than originally envisioned and, in fact, were more costly than a comparable custom-built system. At the same time, several people expressed frustration about the difficulty of communicating to upper management the reasons why COTS-based systems were so expensive to maintain. Anecdotal evidence from these interviews is used to discuss the added sources of maintenance cost. Three different approaches or strategies for system maintenance were observed and are summarized in this article.*

The past 10 to 15 years have seen a strong push within the Department of Defense and other government agencies toward the use of COTS software products in system acquisition. On the surface, this makes a lot of sense – why build something from scratch that already exists, especially if it is a mature product? In fact, with the increasing complexity of today's systems, a total custom system is no longer practical. With the continued use of COTS components in building systems, it is a worthwhile objective to identify sources of cost and approaches to managing them. It is the intent of this article to identify these sources.

As part of an effort to collect data to calibrate a cost model for systems consisting of COTS components, a number of interviews were conducted with project managers, team leads, and other project members maintaining COTS-intensive systems. People consistently told us that these systems were more expensive to maintain than originally estimated and, in fact, were more costly than a comparable custom-built system to maintain. At the same time, we heard frustration expressed about the difficulty of communicating to upper management the reasons why COTS-based systems are so expensive to maintain. If COTS-based systems really are more costly to maintain, what are these additional costs? Are there strategies for managing or minimizing them? These questions are addressed in this article[3].

Before proceeding to answer these questions, we need to define some terms. A COTS software component is defined as it was used in the COCOTS model [2]. This definition contains the following four parts:

1. A COTS component is sold, leased, or licensed for a fee (which includes vendor support in fixing defects if they are found).
2. The source code is unavailable.
3. The component evolves over time as the vendor provides periodic releases of the product (upgrades) containing fixes and new or enhanced functionality.
4. Any given version of a COTS component will reach eventual obsolescence or end of life in which it will no longer be supported by the vendor.

All four parts of this definition have major implications for the added costs of maintaining COTS-intensive systems (compared to comparable custom-developed systems).

At any given time for any given component, there is a choice between upgrading to the next version or doing nothing. There are risks inherent in either strategy. If the first choice is made to upgrade to a new version, there may be unintended interactions with other components, there may be defects introduced as well as unneeded functionality. These types of impacts are discussed in more detail in this article.

If the second choice is made to do nothing, the component will eventually reach end-of-life and will no longer be supported by the vendor. If a problem with the component surfaces at this point in time, the vendor will not fix it and the system maintenance team cannot do much because they do not have access to the source code.

Also, before proceeding, it is important to clarify the type of projects the interviewed managers were involved in. The Software Engineering Institute (SEI) has made a distinction between *COTS-solution* and *COTS-intensive* systems [3]. COTS-solution systems are the typical business or standard information technology systems that are comprised of large application COTS products. Examples include Enterprise Resource Planning applications, human resource, and financial systems. The major COTS component is essentially the system. It provides a user interface, has its own architecture, and has internal business logic that must be followed to be used. On the other hand, COTS-intensive systems are comprised of many COTS components. In these systems, no single component is *king*. There may be many components that handle user interface, data transmission and storage, and data manipulation and transformation. These components interact with each other through custom-developed *glue code* using vendor-provided application program interfaces and with custom-developed application code. The business logic is spread across components and is guided by the way the components are used.

The systems that predominantly made up our sample are mission-critical systems with high reliability and performance requirements and would be classified as COTS-intensive. A number of our projects were air-traffic control systems; we also had ground control systems for missile launches and two ground control systems for satellites. In addition to the high performance and reliability requirements, these systems typically had a large amount of custom application code along with a large number of COTS products (between 10 and 50 was typical).

Two additional points are worth bringing up before discussing the specific sources of added costs for these COTS-intensive systems. We deliberately chose the term *maintenance* rather than the more commonly used term *sustainment* because, like hardware, a COTS-intensive system will, in effect, degrade without dollars and effort spent to manage the impact of multiple components evolving over time. This is the central thesis of this article and is the source of additional cost for these systems.

We do not want to leave the impression that we are against the use of COTS components. Given the complexity of many of today's systems, total custom development is no longer feasible. In addition, the use of COTS components allows system developers to take advantage of the best that the marketplace has to offer and removes the unnecessary *reinvention of the wheel* seen prior to the widespread use of COTS components. Our objective is to help people anticipate and manage the added sources of costs in maintaining COTS-intensive systems.

Major Sources of Added Costs in Maintaining Systems With COTS Software Components

This section discusses the factors that were found to impact the cost of maintaining COTS-intensive systems. Each of the following factors is compared to custom developed systems.

Licensing

The most obvious additional cost burden is component licensing fees. Fees can range from a one-time fee to yearly renewal. The license may be enterprise-wide, site-specific, or per seat (one computer). With one exception, licensing fees did not cause concern among the project members interviewed, presumably because this was an expected, known life-cycle cost. The one exception occurred for a COTS-solution system that was used on a pilot basis at one location. Following a successful pilot, the decision was made to deploy the system worldwide which would entail hundreds of sites. Much to the surprise of the project manager, the per site fee was increased by the vendor. At the time of the interview, he indicated that he assumed that they would get a quantity discount. The price per copy was actually going up. The increase in price was so great that he was seriously considering starting over, this time writing the system themselves from scratch. There are no comparable fees for custom-developed systems.

There is effort required in tracking licensing requirements to ensure that renewals are paid. With different types of licensing and support agreements across different COTS components and vendors, this tracking can become an administrative burden. There is no comparable effort required for custom systems.

Evaluation of New Releases

A major source of cost stems from COTS component volatility.

> Volatility in this case means the frequency with which vendors release new versions of their products and the significance of the changes in those new versions, i.e. minor upgrades versus major new releases. [4]

In contrast to custom-developed code, a COTS software component is controlled by the vendor. The timing and content of releases is at the discretion of the vendor. Major effort may be required to evaluate and understand the implications of upgrading to a new component or perhaps switching to a whole new product entirely.

"Evaluation activities require a test bed that can replicate all deployed system configurations of hardware and software."

COTS software component evaluation addresses the following questions:

1. Are there interactions with other parts of the system?
2. Are there any performance impacts on the system as a whole?
3. Will we need to rewrite *glue code* or application code?
4. Are there any impacts on any custom code?
5. Are there new features that need to be disabled?
6. If there are multiple hardware configurations in the field, can we be sure there are no unintended interactions for any of these?
7. Should we continually upgrade as new versions appear or should we only upgrade for a critical fix?

Evaluation activities require a test bed that can replicate all deployed system configurations of hardware and software. For safety-critical systems, the amount of analysis can be large even though the ultimate decision may be to do nothing. As one person stated, *even when we don't change a version, there is a lot of analysis required. It can be difficult to verify implications with a black box*. The need for this ongoing black-box evaluation is unique to systems with COTS components.

Defect Hunting

Defects appear to be more problematic for COTS-intensive systems than with custom code. After documenting and confirming the existence of a defect, the next step is finding the source within the system. Projects reported that it can be much more difficult with a COTS-based system to pinpoint the source of a problem. It can be difficult to know whether a defect is coming from a COTS component or from other custom developed code. We heard of finger-pointing situations in which a defect was in a COTS product, but the vendor was unable to replicate it because they did not have the same hardware configuration. (Incidentally, this is a problem that may occur during development as well as at any point in the life cycle.) All of this can take time and effort, translating into additional costs.

With a custom system, one can see inside the box. Debugging can follow the path through the code without running into component boundaries. This eliminates finger pointing.

Vendor Support

Vendor support is often used in maintenance to fix defects quickly, provide assistance with the latest product upgrades, or make adjustments to the COTS component in the presence of other product upgrades. The support may range from 24/7 call service to dedicated on-site staffing. If a defect is found and it looks like the source is a COTS component, it is the vendor who must fix the problem (provided the vendor agrees their product has the problem – as noted above, this resolution can take a lot of time and effort). A variety of contractual mechanisms can be in place to guarantee 24/7 support and immediate fixes (this, too, can be a quagmire if the support is unsatisfactory). If the latest release of a COTS software component has new features or interfaces, a vendor's support may be required to integrate a component into the current system. This support may include some tailoring by the vendor to get their component to cooperate with the existing system architecture. Finally, if a vendor has gone out of business, support may be unavailable. Risk-mitigation strategies are discussed later that address this situation.

Upgrade Ripple Effect

After a new version of a COTS component has been evaluated, the installation of the component into the system may have a ripple effect. Due to the new, additional functionality in a component, the system may require changes to custom code, glue code between components, or tailoring of other COTS components. In custom-developed code maintenance, only the fixes and enhancements that are needed are implemented, thus minimizing (but not eliminating) ripple effects.

Hardware Upgrades

People found that upgrades to new software components sometimes required upgrades to new hardware as well. One person noted that vendors were constantly driven to add functionality, putting more demands on the hardware. They have not been able to upgrade the hardware as quickly as they would like.

In a comparable custom maintenance upgrade, hardware performance is considered as part of the upgrade activity. With only the required features implemented, minimal impact to hardware performance can be preserved.

Disabling New Features

There may be new features that need to be disabled for security or performance reasons. The added cost is in the form of additional tailoring of the COTS component. This may require discovering how to disable new features or custom code written to hide or disable the new features. Disabling a feature is not characteristic of custom systems.

Early Maintenance

Because COTS components continue to evolve in the marketplace, it is possible that upgrades may begin before the system is deployed, particularly if the development spans several years. If the components are not upgraded, it is possible that much of the system may have reached *end of life* before the system is even delivered. This was the case according to one of the project managers interviewed; this system had an application base totaling more than one million lines of custom code plus a total of 45 COTS components. Almost half of these components were obsolete by the time the system was deployed.

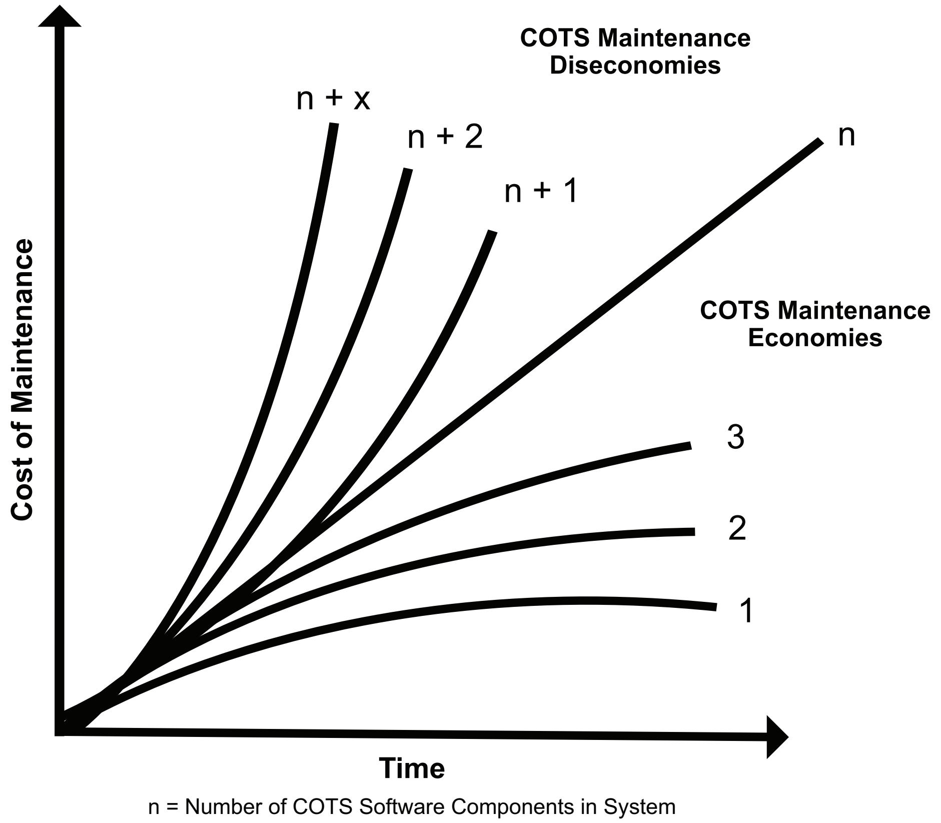
Market Watch

Because COTS vendors can go out of business, a number of those interviewed suggested that a *market watch* be established as a risk mitigation strategy to handle such an event. If a vendor goes out of business, either the component source code or a different component can be purchased. With custom-developed systems, this activity is not required.

Continuous Funding

Another difference between a COTS-based and a custom system is that the systems with COTS components require a more stable funding base. When budgets get tight, funding for maintenance is often sacrificed. With a custom system, enhancement can be delayed until funding is obtained. The consequences of delaying funding with a COTS-based system is that licenses may lapse, bug fixes and upgrades become unavailable, or vendors go out of business with no resources to exercise the risk mitigation identified in a market watch.

Figure 1: *COTS Maintenance Economies Versus Diseconomies*



Number of Components Versus Maintenance Costs

One consistent comment we heard is that the number of COTS components in a system has a strong impact on maintenance costs. A model adapted from one proposed by Chris Abts, called COTS-Life Span Model (LIMO) [5], attempts to explain this phenomenon. The model, depicted in Figure 1, shows two regions divided into maintenance economies (overtime costs go down) and maintenance diseconomies (overtime costs go up). As explained in COTS-LIMO, maintenance costs for a single COTS component go down over time as the experience gained by system maintainers increases, thus improving productivity. The increase in productivity can outpace the increased effort required to maintain the system as the COTS products mature and evolve in divergent directions. However, there is a break-even point with the number of installed COTS software components, (n), where maintenance costs increase disproportionately to the number of COTS products, regardless of the efficiencies gained.

Factors that contribute to the COTS maintenance diseconomies in Figure 1 are those that are discussed in this article. For instance, issues raised with COTS licensing is much more complex with more components. A COTS-intensive system presents multiple licensing strategies, different renewal periods, and different license cost structures. People reported that this can become an administrative nightmare.

Evaluating the impact of upgrades is considerably more burdensome if there are a lot of components (greater than n). The number of possible interactions between components increases exponentially as the number of components increases. When trying to hunt down defects, the complex interactions of many components make the task even more difficult. Configuration management becomes more complex when many components and configurations exist in a system. The possibility of a ripple effect is higher with the impact of component upgrades. There are more unwanted features with more components. The market watch becomes a large-scale activity.

The idea of this model was verified when we kept hearing that the complexity of maintaining a COTS-based system increases dramatically as the number of different COTS components increases. There is much more potential for interaction as well as more potential upgrades that have to be examined.

Three Risk-Mitigation Strategies to Deal With the Challenges of Maintaining COTS-Intensive Systems

This article has discussed sources of additional cost in maintaining COTS-intensive systems. Across the projects interviewed, three strategies for dealing with COTS volatility were observed. Each of these strategies is discussed next.

Revert Back to Source Code

Several of the projects interviewed opted to maintain one or more *critical* COTS components themselves. In one case, the product (an operating system) was allowed to reach end of life and the project purchased the source code from the vendor. From that point on, they no longer had vendor support but were able to make fixes themselves. This decision was made because it avoided the necessity for hardware upgrades. It removed the risk of being unable to fix future problems. Alternatively, several other projects replaced critical COTS components with their own custom-developed software.

This strategy places control for fixing problems back in the hands of the maintenance organization. A downside is the additional expense (purchasing the source code or developing it from scratch). In the case given dealing with the operating system, this strategy was part of a larger strategy to freeze the hardware configuration, much of which was special purpose, for a period of time until the next generation system could be deployed.

Divide and Conquer

This strategy divides the COTS software components into two categories: non-critical and critical. The non-critical COTS components are not upgraded. Resources are focused on the set of critical components. For these components, market watch and evaluation activities occurred and the decision to upgrade was made individually for each critical component.

This strategy is driven by the need to balance the ongoing costs required for maintaining a COTS-intensive system with limited resources. The upside of this strategy is that it saves money by ignoring a subset of components. The downside of this strategy is that a portion of the system remains stagnant and unsupported.

Design for Change

The third strategy uses *information hiding* in the form of wrappers to protect the system from unintended negative impacts of multiple component upgrades. One interviewee said when describing this strategy that they wanted to be able to replace a product without damage to the rest of the system. As an example, they had a wrapper around the database. It could be a flat file or relational database – the custom application didn't care. This strategy requires more thought and effort up front. The project in our sample that used this strategy had a strong project sponsor right from the beginning who argued successfully for additional resources to design for change from the beginning. This was a project that was planned for a long life with safety-critical requirements.

"The sources of added costs discussed ... were identified through anecdotal evidence obtained from interviews. The next step is to quantify these sources and the parameters that impact each. We are looking for opportunities to continue this investigation."

The advantages of this strategy are clear: There is much more assurance against unintended ripple effects from upgrades or even product replacement with a product from another vendor. The disadvantage is the necessity for resources early in system development when the typical focus is on getting the system deployed rather than worrying a great deal about the life-cycle consequences of decisions.

Concluding Remarks

This article has discussed the sources of additional costs required to maintain COTS-intensive systems. As noted in the introduction, we do not want to leave the impression that we are against the use of COTS components. One of the people interviewed expressed the view that the continual evolution and maturation of COTS components is, in fact, one of the real positives of using commercial components in a system.

It is the authors' objective to help people understand some of the added sources of costs in maintaining a COTS-intensive system, particularly by bringing attention to areas that may not have been anticipated. In particular, projects should understand the life-cycle implications of integrating a large number of COTS components. More thought should be given early to the impact of upgrades on the entire system, the reliance on vendors to fix problems, and the strategies that will be used in dealing with multiple products, each evolving at the discretion of the vendor. These concerns become especially problematic with high assurance, high performance systems.

The sources of added costs discussed in this article were identified through anecdotal evidence obtained from interviews. The next step is to quantify these sources and the parameters that impact each. We are looking for opportunities to continue this investigation.◆

References

1. Reifer, D.J., V.R. Basili, B.W. Boehm, and B. Clark. "COTS-Based Systems – Twelve Lessons Learned about Maintenance." COTS-Based Software Systems, Third International Conference, ICCBSS 2004, Redondo Beach, CA.
2. Center for Systems and Software Engineering. COCOTS <http://sunset.usc.edu/research/COCOTS/index.html>.
3. Oberndorf, Tricia, Lisa Brownsword, and Carole Sledge. "An Activity Framework for COTS-Based Systems." Carnegie Mellon University (CMU)/SEI-2000-TR-010. Pittsburgh: SEI, CMU, 2000.
4. Abts, Chris. "COTS-Based Systems and Make vs. Buy Decisions: The Emerging Picture." International Workshop on Reuse Economics, Austin, TX. 16 Apr. 2002 <www.

favaro.net/john/icsr7/papers.htm>.
5. Abts, Chris. "COTS-Based Systems Functional Density – A Heuristic for Better CBS Design." Proc. First International Conference on COTS-Based Software Systems, Feb 2002.

Notes

1. This model, named Constructive COTS Model (COCOTS), is one of the COCOMO family of models.
2. The interviews and model calibration were sponsored by the Federal Aviation Administration's (FAA) Software Engineering Resource Center. The interviews were conducted by Dr. Chris Abts and Dr. Betsy Clark.
3. For a discussion of lessons learned in maintaining COTS-intensive systems, see Reifer, et al. [1].

About the Authors

Betsy Clark, Ph.D., has been involved in the practical application of measurement for predicting, controlling and improving software process and product quality since 1979. She is the president of Software Metrics, Inc., a Virginia-based consulting company she co-founded in 1983. Clark is a primary contributor to *Practical Software Measurement: A Guide to Objective Program Insight*. She was also a principle contributor to the SEI's core measures. She has contributed to numerous studies of software best practices for the DoD and FAA. Clark is a research associate at the Center for Systems and Software Engineering at the University of Southern California. She worked with Barry Boehm and Chris Abts to develop and calibrate a cost-estimation model for COTS-intensive systems under sponsorship of the FAA. She is currently supporting the U.S. Customs and Border Protection's Cargo Systems Program Office, working to implement performance measures within the framework of the Office of Management and Budget's Performance Reference Model. Clark has a bachelor of arts in psychology with distinction from Stanford University and a doctorate in cognitive psychology from the University of California, Berkeley. She is also an accomplished equestrian, having earned a Gold Medal from the United States Dressage Federation.

Software Metrics, Inc.
4345 High Ridge RD
Haymarket, VA 20169
Phone: (703) 754-0115
Fax: (703) 754-3446
E-mail: betsy@software-metrics.com

Brad Clark, Ph.D., is an independent consultant in the area of software measurement with 12 years experience in software development and management best practices. He is vice-president of Software Metrics, Inc., and specializes in the area of software cost and schedule risk analysis. Clark is a research associate with the Center for Systems and Software Engineering at the University of Southern California. He has co-authored the book *Software Cost Estimation With COCOMO II* with Barry Boehm and others. Clark helped define the COCOMO II model, collected and analyzed data, and calibrated the model. He has a bachelor's degree in computer and information science from the University of Florida, a master's degree in software engineering, and a doctorate in computer science from the University of Southern California. He is a former Navy A-6 pilot.

Software Metrics, Inc.
4345 High Ridge RD
Haymarket, VA 20169
Phone: (703) 754-0115
Fax: (703) 754-3446
E-mail: brad@software-metrics.com

Coming Events

July 9-11
CBSE 2007 10th International ACM SIGSOFT Symposium on Component – Based Software Engineering
Boston, MA
www.csse.monash.edu.au/~hws/CBSE10

July 9-11
SEDE 2007
16th International Conference on Software Engineering and Data Engineering
Las Vegas, NV
http://sede07.cs.uh.edu

July 9-11
SEKE 2007
19th International Conference on Software Engineering and Knowledge Engineering
Boston, MA
www.ksi.edu/seke/seke07.html

July 9-12
SETP 2007
International Conference on Software Engineering Theory and Practice
Orlando, FL
www.promoteresearch.org/2007/setp/index.html

July 15-18
SCSC 2007 Summer Computer Simulation Conference
San Diego, CA
www.sce.carleton.ca/faculty/wainer/SCSC07/SCSC'07.htm

2008

STC Systems & Software Technology Conference

Systems and Software Technology Conference
www.sstc-online.org

Coming Events: **Please submit coming events that are of interest to our readers at least 90 days before registration. E-mail announcements to: nicole.kentta@hill.af.mil.**