Selection of Components for OTS Component-based Systems

Brad Kizzort
Harris Technical Services Corporation
1225 Evans Road
Melbourne, FL 32904
(321) 723-3999
bkizzort@harris.com

Abstract—Acquisition of software-intensive computer systems shifted toward the inclusion of Off-the-Shelf (OTS) components in the 1990s in order to control spiraling software costs. The initial results of this shift are in, and there have been significant lessons learned along the way. This paper examines some of the key attributes for success and failure in OTS-based development programs including requirements management and component selection. The paper recommends criteria for selection of components based on the system characteristics.

The paper discusses tradeoffs between commercial components, open source components, and legacy components and how program characteristics, e.g. program lifetime, development schedule, and maintenance plan, affect component selection. Also included is discussion of how direct project experience compares with the project experience reported in recent papers.

TABLE OF CONTENTS

1. INTRODUCTION
2. MANAGING REQUIREMENTS
3. ARCHITECTURAL SUPPORT FOR EXTENSIONS
4. TESTING
5. INSTRUMENTATION
6. MAINTAINING THE SYSTEM
7. CONCLUSIONS
8. REFERENCES
9. BIOGRAPHY

1. INTRODUCTION

Government and commercial systems procurement in the 1990s turned to Off-The-Shelf (OTS) software components to reduce system costs and shorten delivery schedules. There were successes and failures in these initial programs and numerous lessons have been learned in trying to adapt and use OTS software.

Definitions

First, a few terms will be defined to clarify the nomenclature used in this paper:

- Commercial OTS (COTS) software refers to software that is developed and sold by software vendors to multiple customers and is maintained by the original vendor
- Government OTS (GOTS) refers to software that was developed for the government and can be re-used without licensing costs for other government projects. Maintenance of the software baseline is provided by the supplying organization
- Non-Development Item (NDI) refers to software that was previously developed, requires no software licensing cost, and is being reused with little or no modification for a new project. Maintenance must be provided by the using organization

Particularly in the government software market and in small market niches, there are vendors that re-use and modify software for specific programs. The software source baseline is taken from a previous project and is configuration managed as a separate branch for a new program. Although this type of software is frequently identified by the vendor as COTS, there is no common baseline that is maintained and updated for all customers. Therefore it is classified here as NDI, since all maintenance and updates must be provided or contracted for by the using organization. The NDI category also contains open-source components and systems for similar reasons.

- A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [1].

The definition provided for a software component comes from the field of component-based software engineering. The usage of the term in that field is frequently restricted to components defined and built using one of the current distributed object technologies, e.g. CORBA, DCOM, or...
JavaBeans, however, this paper also considers large-scale components, e.g. Microsoft Word, OS/COMET, etc., that can be adapted and incorporated into larger systems.

Focus

This paper discusses primarily the use and selection of COTS software components, but many of the criteria used for the selection of COTS components are applicable to GOTS and NDI as well. The distinguishing differences are in licensing and maintenance/upgrade responsibilities, but there may also be differences in reliability, breadth of applicability, etc.

Using COTS software is not new. Commercial operating systems and language compilers have been in use in commercial and government systems for decades. COTS Relational Database Management Systems (RDBMS) are used as a component in many systems. Callable libraries provide functionality for statistics, mathematics, indexed file access, etc. Developers and maintainers of systems using these COTS products are accustomed to working around the bugs discovered in these components during the development cycle. Bugs introduced and discovered when applying updates for maintenance cause the update to be held until the vendor can provide a fix or another workaround can be found. Information Systems departments frequently put a new version of an operating system on a test bed for months prior to rolling it out to the entire network to prevent rolling out undiscovered bugs in the COTS product.

As more of the system functionality is provided by COTS software, the “development” of the system shifts from a creative activity to a compositional activity [2]. The system is pieced together based on the composers' knowledge of product offerings and progressive refinement of the general system requirements. It shifts away from the requirements analysis, system design, decomposition into subsystems, development, integration, and test cycles of conventional system development. COTS-based system composition versus conventional system development could be represented as a sliding scale rather than two distinctly different approaches.

2. MANAGING REQUIREMENTS

“Have some wine,” the March Hare said in an encouraging tone.

Alice looked all round the table, but there was nothing on it but tea. “I don't see any wine,” she remarked.

“There isn't any,” said the March Hare.

“Then it wasn't very civil of you to offer it,” said Alice angrily. ²

The Catch-22 of COTS-based development is that selecting the COTS products requires knowing the system requirements but the optimal requirement set may be affected by what is available from existing COTS. One of the keys to managing costs by using COTS components is managing the system requirements such that the majority of functionality can be purchased or obtained off-the-shelf. As the system requirements increasingly stray from what is available, the cost of a full compliance increases. Adding more COTS components to achieve compliance is not necessarily the answer, because, as the number of COTS components in the system increases, the integration and maintenance costs usually increase.

As Barry Boehm has said, “According to the old process, system requirements drove capabilities. In the new process, capabilities will drive system requirements … it is not a requirement if you cannot afford it.” [3]

What makes this system unique?

One of the questions the system engineer must ask himself about a new system is: “What makes this system unique?” Identifying what sets the system apart from previous systems focuses on the requirements that prevent a solution from being procured totally off-the-shelf.

For example, elements that might require customization for a satellite ground control system are:

1. Unique spacecraft bus
2. Unique payload
3. Inter-satellite networking and constellation mission coordination
4. High precision attitude/orbital knowledge or control
5. Highly automated operations with built-in ground system and/or onboard decisions.

If a satellite ground control system is going to control a single satellite with a common commercial spacecraft bus, then there are several vendors that can provide a nearly complete COTS solution for the ground system. Each vendor may take a slightly different approach in their control systems, but they will cover the basic requirements for monitoring and control of a spacecraft. Issuing the requirements document to vendors during the request-for-information (RFI) phase, with a request to identify the areas requiring development rather than striving for 100% compliance is one way to identify requirements that will drive the cost of custom system development. Requirements that are identified as requiring development by one or more of the vendors and that are not related to unique system elements are probably not requirements at all and should be scrutinized closely.

Prototyping to verify requirements

An ideal way for developers and system integrators to become familiar with a COTS product is via rapid prototyping using the COTS. Trading off between the

² Alice’s Adventures in Wonderland, Lewis Carroll.
system requirements, architecture, and COTS capabilities is an essential activity of successful COTS-based development [2, 4], and a system prototype is a good environment in which to make those trades. Several organizations have reported that evaluation of the COTS products within the system context is necessary to select the right products [5, 6]. In order to benefit from a prototype, the requirements effort needs to start with general requirements that are refined during the prototyping effort. COTS evaluation and COTS familiarization are accomplished during the prototyping, and requirements are refined in parallel with the prototype. If a waterfall-model requirements effort is initiated to prepare for the prototype, then the results are likely to hamper the prototyping effort rather than aid in selecting the architecture and COTS [7].

In this respect, a research organization that launches multiple small programs has an advantage over a large single program organization. Although the large program can use rapid prototyping and spiral development to converge on its requirement set, the investment involved in learning the COTS products makes it more cost-effective to quickly select the COTS products for the program and move into development. The multi-program organization can use several COTS products and phase-out and phase-in products over time as technology changes or better products become available. The cost of learning can be leveraged over several programs.

3. ARCHITECTURAL SUPPORT FOR EXTENSIONS

"Who are YOU?" said the Caterpillar.

This was not an encouraging opening for a conversation. Alice replied, rather shyly, "I--I hardly know, sir, just at present--at least I know who I WAS when I got up this morning, but I think I must have been changed several times since then."3

Figure 1 COTS Models

The architecture of the system being built must support adaptation and extension of the COTS products for a COTS-based development program. If large-scale COTS components are used, then the COTS components must be adaptable and extensible. The COTS products must support this adaptation without modifying the source code baseline of the product, although it may include source code development through dynamically-linked libraries, scripting, access to distributed components, providing a callable Application Programmer Interface (API) or a combination of all of these.

COTS components can be divided into two groups: conventional API and frameworks. Figure 1 is similar to diagrams others [8] have drawn and shows the difference graphically. There are some COTS products that have elements of both groups, e.g. Object-Oriented Graphical User Interface (GUI) libraries, RDBMS’s. In practice, the configuration management issues and maintenance issues for both groups are similar, but the approach to extension and adaptation is quite different. Whichever path is chosen, the key selection criterion is how easily the COTS product supports adaptation and extension within the system context and in the areas of the system that are likely to change.

Developing with Frameworks

Frameworks may either provide generic services to support an application, specialized services within an application domain, or both. The framework is usually functional without any user customization or tailoring. This allows the user to incrementally customize the framework for the application, making rapid prototyping in a framework environment truly rapid.

Extending with plug-ins—A plug-in is a separately developed module that complies with a plug-in interface defined by the framework. The framework makes calls to the plug-in at specified points in processing, or as a result of configuration files or scripting. The advantage of a framework that supports plug-ins is that plug-ins provide a way to add extensions that run at the speed of compiled code.

---

3 Alice’s Adventures in Wonderland, Lewis Carroll.
The concept of plug-ins is extended to distributed frameworks such as DCOM, CORBA, or Java RMI. In these cases, there is a call that is extended into another software process space either on the same computer or across the network. Compared to an in-process plug-in, there is a longer latency, but functionally the work is conducted in the plug-in linked at run-time.

These distributed frameworks can be part of an architecture that supports COTS-based development and they certainly make distributed application development much easier. However, like an RDBMS, the services provided are very generic and they typically don’t provide any application-specific functionality without additional source code or other COTS components. They are infrastructure COTS components that support distributed architectures.

Support of plug-ins is not required attribute of a COTS component, but it is highly desirable, especially when no COTS product is available that supports the desired application features, so that software development is required beyond the creation of wrapper classes and glue code for integrating the COTS components.

**Data-Driven Operations**—Data-driven software may use configuration data in several forms, ASCII configuration files, binary configuration files, indexed files, or relational tables. This information may be referenced only at startup, loaded when needed, and/or reloaded on request. A simple example of this is a resource file that configures an Xwindows/Motif application. Another example is the set of telemetry definition files that define processing and commutation formats for a telemetry processing system. The configuration information can control many aspects of the COTS product, e.g. GUI appearance, internal processing, logging.

This is a very important method of adaptation that most COTS products support. The key data-driven operations criterion for selection of a COTS product is whether the method of access to the configuration data supports the intended usage within the end system. For instance, a satellite fleet management system should be able to support changing the command and telemetry definitions for one of the satellites without affecting or restarting processes for other satellites in the fleet.

If the configuration information for a product is accessed at run-time, then a mission-critical system must also support reliable access to the configuration data, i.e. a high-availability Network File System (NFS) server if configuration data is stored in shared files or a high-availability RDBMS if configuration data is stored in relational tables. This capability may place requirements for additional COTS components in the integrated system, such as RAID disk arrays, clustering software, and/or journaled file systems.

**Scripting**—In a large-scale component, a scripting language is a critical feature for adaptation and enhancement. Microsoft saw it as important enough to replace and standardize the scripting language across its suite of Office automation products, so that they all now use Visual Basic for Applications as a built-in scripting language. Rational Software used this customization capability extensively in its RequisitePro product. The reason scripting provides such a powerful capability is that the domain-specific capabilities of the product, in this case the text editing and formatting capabilities of Word, are made available to the application developer and frequently even the end-user as statements in the scripting language. The developer can then work with higher level functionality in an interpretive-style language, seamlessly using and applying the capabilities of the large-scale component.

Not surprisingly, many of the scripting languages incorporated into COTS products are proprietary languages, since the language supports domain-specific features in addition to typical programming language features. There is a dilemma for the vendors between keeping the language easily accessible to non-programmers and making the language more rigorously structured such that third-party development tools, e.g. integrated development environments, language-sensitive editors, can be used. There are a few COTS products that use open-source scripting languages, like Tcl or Python, but even here the third-party development tools available may not support the domain-specific features very well.

Some features of the scripting language that previous projects have found very useful are that it:

1. Has an ASCII representation that supports standard programming environment tools such as text editors, source code control systems (configuration management), and grep
2. Is itself extensible by the application developer through application-defined functions (plug-ins), or other mechanisms
3. Has a debugging capability that includes pausing, single-stepping, and display of variables.

**Inheritance**—An object-oriented framework will support extension and adaptation through inheritance. With modern object-oriented standards, inheritance can be used by developers even without access to the source code of the product. Due to the more complete understanding of the framework required for this type of extension, it is typically only used by software developers during application rather than the data-driven and scripting extensions that are usually accessible to the end user [8].

There are a number of object-oriented (OO) framework products particularly for GUI development. The usability of an OO framework is related to how understandable the abstractions created by the vendor are. Thus, prototyping
with the COTS candidates is particularly important to the selection of an OO framework.

**Developing with the conventional model**

Developing with conventional COTS components rather than a framework typically means writing application wrappers for components to standardize interfaces, developing main programs that call COTS library functions, and writing glue code to adapt data and/or control outputs of one component to the inputs of other components. Skipping the writing of wrappers may speed the application development in some cases, but will hamper maintenance later in the system development cycle [9, 10].

Because developing with conventional COTS components is more code-intensive than a framework, prototyping is not as rapid. It is still useful to prototype for familiarization with the COTS components and confirmation of key capabilities of the components.

**Access to Data**

Small components typically have clearly defined inputs and outputs, and internal data formats can be ignored by the application developer/integrator, since they are not necessary to interface with other components. However, large-scale components may have many data storage formats and may restrict access to a proprietary set of tools. This is a key selection criterion for large-scale COTS components [3]. The data used and produced by the component must be either defined and accessible via standard operating system calls, or a set of utilities / application interfaces must be provided by the vendor to access the data. Without access to the data, the integrator will be hamstrung in integrating other components, e.g. additional data visualization tools, as well as in maintaining the system by adding or replacing components in the future. Web-based access to data, via HyperText Transfer Protocol (HTTP) and Secure Sockets Layer (SSL), is a rapidly growing option that is attractive as long as it meets the performance and security requirements for the system.

**Examples of Developing with Frameworks**

The Rational RequisitePro tool is an example of a product that took advantage of the framework that Microsoft created in its Windows operating systems and Office product suite. Through the use of Visual Basic scripting and plug-ins within Microsoft Word and COM components to interface with the database, a product was developed that used the editing and formatting capabilities already in Word and interfaced to an RDBMS to hold the requirements. As might have been expected, there were some initial problems with stability, as Word was stretched in areas that it had not been stretched before, but the result was a useful requirements tool with a very familiar editing interface for many engineers. Because the framework is still open and accessible, the RequisitePro tool can be further extended by the end user.

Implementations of CORBA, e.g. Iona’s Orbix and TAO, simplify the development of network interfaced objects, allowing developers to rapidly develop distributed applications. It is, in a sense, a framework, but with very limited functionality until objects are developed. The Object Management Group has also defined services and domain facilities using the CORBA interface specifications, which can turn it into a development framework as defined here. In project experience, the ease of developing network applications may lull developers into not adequately considering distributed application issues, such as deadlock, network load, latency, throughput, versioning, and object-to-object interactions through a broker object. These issues can also arise during integration of COTS components with CORBA interfaces, since the COTS developers will make usage and interaction assumptions about product components that may not fit the system design. Modeling and simulation tools developed for network applications or prototyping with simulated system conditions can be useful in identifying problems early in the program.

4. **TESTING**

She ate a little bit, and said anxiously to herself, “Which way? Which way?” holding her hand on the top of her head to feel which way it was growing, and she was quite surprised to find that she remained the same size. To be sure, this generally happens when one eats cake, but Alice had got so much into the way of expecting nothing but out-of-the-way things to happen, that it seemed quite dull and stupid for life to go on in the common way.4

One of the primary advantages touted for COTS-based systems is that the user base for a COTS product insures that its functionality has been thoroughly debugged in alpha, beta, and end user testing. From practical experience, e.g. desktop operating system and office automation products, this is not a big advantage. As the application of a product strays from its design and common usage, new paths are executed that may uncover new errors. As bug fixes and enhancements are included in new releases, new interaction errors may occur in a system based on the product, even though the product passes its own regression tests. The level of the impact on the system integrator is typically dependent on the maturity of the product, although the development and testing processes of the vendor also affect the impact.

Given these problems, the tests developed for accepting and regression testing of the system cannot rely solely on product vendor regression tests for critical function areas. There may be system requirements that are traced solely to a specific COTS product and do not involve interactions with other products or integration software, but these will be a

---

4 Alice’s Adventures in Wonderland, Lewis Carroll.
minority fraction of the system requirements. The best path

to follow here is the traditional mapping of system
requirements to system test procedures without regard to a
specific COTS product’s capabilities. The COTS product
vendor is responsible for testing the COTS product
capabilities, which may be much broader than the
capabilities required by the system and those generic
capabilities do not need to be re-tested at the system level.
Saving system test time by relying on the vendor’s
regression tests for the subset of capabilities being used in
the system is not a prudent path.

5. INSTRUMENTATION

Alice had been looking over his shoulder with
some curiosity. “What a funny watch!” she
remarked. “It tells the day of the month, and
doesn’t tell what o’clock it is.”

Instrumentation of a system plays a critical role in
monitoring and troubleshooting a system [11]. The
information provided by system instrumentation is vital to
troubleshooting during the initial integration as well as
during the maintenance phase as new releases of
components are integrated into the system. Determining
which component is at fault without any information on the
internal operations of the components adds to the
complexity of troubleshooting.

The difficulty with COTS-based systems is that COTS
products vary widely in the amount of instrumentation built
into the product. Frequently, the integrator is forced to
depend solely on instrumentation built into the wrappers and
glue code and on tools available from the platform vendor.
Some commercial products do include current status
information, process trace logs, and action/response logs.
To some extent this reflects the maturity of the COTS
component, e.g. operating systems and RDBMS products
have long been used as COTS components and usually
provide the instrumentation necessary to monitor
performance and troubleshoot problems. Presence of built-in
instrumentation should be a selection criterion in the
COTS component.

6. MAINTAINING THE SYSTEM

“Yes, that’s it,” said the Hatter with a sigh: “it’s
always tea-time, and we’ve no time to wash the
things between whiles.”

The impact of volatility in the COTS components of a
system begins during the COTS selection process and does
not end until either the end of system life or when the
system baseline is frozen. This impact is also the primary
reason that many researchers question whether the use of
COTS components will reduce total lifecycle cost [13].
Since the operating system (OS) is a key COTS software component that often drives upgrades of other software components and is driven by the computer hardware evolution, Figure 2 illustrates the lifecycle of several releases of the Solaris OS. Sun’s vintage support policy, established in May 2000, set two key milestones in the OS life. Vintage support begins two years after the last ship date for a version of the OS. During the first two years of vintage support, the standard support rate is charged and no cosmetic bugs or enhancements are considered for the OS. During the last three years of vintage support, access to new patches (still no cosmetic bugs or enhancements) requires a premium support rate. Five years after the last ship date, any support is a custom quote from Sun. Solaris was chosen as an example, but it is expected that other computer vendors will have similar policies.

Frozen COTS and Fluid COTS

One promise a COTS based system can fulfill is a shorter development cycle, though not all procurements achieve this [7]. A custom system software development of the complex systems being attempted today can take longer to build than the commercial life of the hardware that hosts it. Therefore, COTS products will continue to increase their roles in new system development. The COTS product selection process needs to take into consideration whether the system will use a frozen baseline once it goes operational or a fluid baseline that is updated periodically with new hardware and software.

Selection of COTS for a fluid baseline requires an emphasis on the support, maintenance, and compatibility policies of the vendor. This is where NDI software can be less attractive than COTS, since it is maintained by the using organization. The using organization has to retain personnel with expertise in maintaining the NDI software, including bringing it up to date with new operating systems and hardware platforms. Conversely, the organization has more control over changes in the NDI software.

Optimizing the Number of Components

System architects for a COTS based system are faced with a dilemma. Using a few large-scale components minimizes the problems with COTS interdependencies and should reduce integration time and maintenance. Integrating many small components maximizes the flexibility in satisfying requirements and replacing components as the system evolves [12]. Each COTS product may have dependencies on other COTS products. Typically, each will at least be

\[
\begin{align*}
\text{First Release to Last Ship Date} & \\
\text{Vintage Support, Standard rate, no cosmetic patches, no enhancements} & \\
\text{Vintage Support, Premium Rate, no cosmetic patches, no enhancements}
\end{align*}
\]

Figure 2 Recent Solaris Releases

Figure 2 also illustrates that to span a 15-year system life with a commercial OS is going to require either a frozen system approach or periodic updates to the system. As an older example of a frozen system approach, approximately thirty-three years ago there was a military system that was built using IBM 360 mainframes. A COTS operating system was used and modified for the real-time application and the software baseline was frozen. Twenty-one years later the system was still operational, using IBM 4341’s running in 360 compatibility-mode. At that time there was a maintenance problem in that the disk drives were failing and there were no longer any spare disk drives available that the frozen operating system would recognize. System designers today will probably have to handle a similar problem in five years instead of twenty.
dependent on the operating system release. These interdependencies complicate system integration and system upgrades, e.g. product A version 2 needs OS version 2.7, but product B for OS version 2.7 is not available, yet. The system architects must optimize the number of components based on the integration schedule and maintenance plan for the system.

Frequently, a large-scale component will contain integrated COTS components internally. To the extent that this integration is seamless, it protects the user from dealing with the interdependencies within that set of products during maintenance. The problems still exist, but they are managed by the component vendor rather than the user. This could result in updates to the product for a new operating system release that require more time by the vendor for integration and testing within the vendor maintenance cycle. The vendor’s update history is a component selection criterion particularly when the system is using a fluid baseline approach.

One element of the complexity of delivery dependencies for the system integrator is not specific to COTS software. In fact it is exacerbated by software that is not truly off-the-shelf. In the author’s experience and experience reported by others [7], dependence on NDI or COTS software, which has not been completed can be a significant source of schedule delays and integration difficulty. Even if the originator of the software meets the delivery schedule, the immaturity of the component may extend integration due to additional patch releases of the component occurring during system integration. It may seem obvious to state that the existence of the COTS component is an important criterion for COTS selection, but many programs are undertaken with a dependency on COTS or NDI components or features that are still under development. The system integrator must recognize this as a source of risk and should include it in the selection factors for the COTS components.

7. CONCLUSIONS

Increasing system complexities and accelerating technology advances will assure that COTS products will continue to increase their role in system development in the future. Ongoing research in evaluation techniques, comparative costs, cost estimation [13], successful integration approaches, and successful development organization will enhance the ability to build and maintain COTS-based systems. One of the key successful project attributes identified here and by others is an iterative requirements development approach that trades off between product selection, architecture, and system requirements. Selection criteria discussed here include:

1. Support for user extension and tailoring
2. Access to data
3. Visibility into operations through instrumentation
4. Vendor update history

8. REFERENCES


9. BIOGRAPHY

The author, Brad Kizzort, has over 20 years of experience in software engineering with several companies. Over that period, he has worked on system development projects and on the software products and reusable software components of those organizations. His current employer, Harris Technical Services Corporation, offers a suite of command and control products under the OS/COMET trademark. As a systems engineer in that product group he strives to make the products as useful and effective as possible for COTS-based system integrators.