Software Reuse and Commercial Off-the-Shelf Software

By Dan Galorath
President
Galorath Incorporated
El Segundo, CA

The most radical possible solution for constructing software is not to construct it at all.
Fred Brooks

Introduction

Organizations faced with the difficulties and costs associated with the development of software have turned to the reuse of existing software or using commercial off-the-shelf (COTS) software as an option. Reuse, whether involving home-grown or COTS components, certainly promises lower cost, better quality, a decrease in risk, and the potential for a less stressful development process. Many such efforts succeed, but the promises of decreased cost and risk are not always realized. Requirements, algorithms, functions, business rules, architecture, source code, test cases, input data, and scripts can all be reused. Architecture is a key for reuse.[1]

Many programs that plan substantial reuse find that the assumptions made concerning how much functionality could be achieved were overly optimistic. They are then disappointed when the amount is less than projected or they experience much higher costs for reuse than had been estimated. Reuse is not a panacea, the ultimate saver of schedule, or cost reduction measure that optimistic estimators or zealous managers promise. In reality, reuse or COTS can lower cost, but only partially. COTS application software often satisfies less than 40 percent of the functionality of an application.

Even when functional requirements are reasonably well satisfied, critical nonfunctional requirements such as security, reliability, and performance must be addressed, resulting in schedule and cost impacts. If the functional or interface requirements are not satisfied, wrappers (additional code required to make the new development able to use the existing software) must be planned, designed, developed, and tested.

In all cases, the system or software architecture must be sufficiently mature to allow the detailed design of critical interfaces and the conduct of reasonable trade-offs to enable the evaluation, selection, acquisition, and integration of the capability into the system or software architecture. Only when components are produced like hardware chips, that is, components that are designed for reuse, include appropriate inputs and outputs, and have been fully tested for the environment can the risk of reusing someone else’s code be reduced.

When working with embedded systems (software embedded with hardware), critical system, hardware, and operational considerations greatly complicate the evaluation trade-offs and the selection process by forcing the analysis to address external considerations and usability factors at the same depth as internal

system and software considerations. The decision to use COTS or reuse a legacy component cannot be made simply because the items "fit in the architecture." The use must be based on a certainty that the components will prove operationally sound across the full range of operational scenarios they must support.

Often reuse decisions are made by defining high level views of what might be needed and identifying off-the-shelf components, preexisting functional designs, and reuse components that roughly satisfy the identified requirement from catalogs or vendors' cost sheets. These decisions then make their way into the estimate only to be reversed later — above the projected cost and outside the projected schedule.

When discussing reuse, it is important to identify the type of reuse. Most reuse falls into one of the following categories: incidental reuse, planned reuse, incremental capability, or COTS.

***Incidental reuse** — The most common form of reuse involves an attempt to use software developed for one purpose or application in a new application. This approach is far from cost-free. The software must be partially redesigned, reimplemented, and then retested to ensure it does what it should do, and does not do what it should not do. This can be a potential minefield that can cause an organization to inherit all the problems of the preexisting software and reap few of the benefits. Reuse may actually cost more than developing new software because of the poor state or lack of fit of the reused components. Many managers, when planning for software reuse, forget that the reuse software must be tested in the new environment.

Planned reuse — This involves software developed with reuse as a goal during its development. Developers spent extra effort to ensure it would be reusable within the intended domains. The additional cost during development may have been significant but the resulting product can achieve dramatic savings over a number of projects. The SEER-SEM estimation model shows that the additional costs of building software designed for reuse can be up to 63 percent more than building with no consideration for reusability.

Incremental capability — This is the addition of functionality to an existing system, whether through upgrade or incremental deliveries of a system under development. The analysis required is identical to those used for other planned and unplanned reuse; the product is an additional capability added to the existing system.

COTS — The COTS term is applied to almost all retail software. COTS components can be anything from an operating system to a word processor, a language compiler, or a component that is invisibly integrated into a software program.

Many like to think of reuse as a silver bullet. In fact, reuse can be
Pandora's box if inappropriate assumptions are made about the applicability of the software to be reused or if the reused software has inherent problems.

Reusable Software

Table 8.1 outlines the characteristic differences among types of reusable software.

The effort required for reuse of existing software depends on several factors that must be well understood before a determination to reuse functionality is made. An efficient approach is to convert the preexisting software into an effective (equivalent) number of size units (lines, function points, or other units) using formulas developed from experience.

Table 8.1 Comparison of Types of Reusable Software

	COTS	GOTS	Planned Reuse	Incidental Reuse
Ready to use and documented	Yes	Sometimes	Often	Sometimes
Allows programs to offset rising development costs	Often	Often	Often	Often
Tends to follow open standards, making integration easier	Often	Sometimes	Sometimes	Occasionally
Designed for reuse, generalized and well tested	Usually	Often	Sometimes	Occasionally
Often updated and improved	Usually, due to competitive pressure	Occasionally	Sometimes	Seldom

Reuse involves three activities, each of which has a price: redesign, reimplementation, and retesting. Redesign arises because the existing functionality may not be exactly suited to the new task; it likely will require some rework to support new functions, and will likely require reverse engineering to reveal its current operation. Some design changes may be in order. This will result also in reimplementation, which generally takes the form of coding changes. Whether or not redesign and reimplementation are needed, plan to conduct some retesting to be sure the preexisting software operates properly in its new environment.

The effective size of the existing software can be determined using the formula:[2] Effective size = existing size · (0.4 · redesign % + 0.25 · reimplementation % + 0.35 · retest %)

The various redesign, reimplementation, and retest components can be estimated by breaking each one down into its several components, then using some additional formulas gained from experience. Table 8.2 shows the components and formulas. The procedure should be tailored for an organization's process specifics. For example, the five components of redesign (as shown in the table) are: (A) architectural design change, (B) detailed design change, (C) reverse engineering required, (D) redocumentation required, and (E) revalidation required.

Table 8.3 illustrates a calculation of reimplementation based on 30 percent recoding required, 32 percent code review, and 35 percent unit testing.

After using the formula provided in Table 8.2 to compute the redesign, reimplementation, and retest percentages, the following equation is used to compute effective size:[2]

Effective size = new code + preexisting code · (0.4 · redesign % + 25 ·reimplementation % + 35 · retest %)

For example, if there are 750 preexisting function points and the redesignis 15 percent, eimplementation is 10 percent, and retest is 18 percent, the following formula would be used to determine effective function point rating and the result would be 116 effective function points.

$$750 \cdot (0.4 \cdot 15\,\% + 25 \cdot 10\% + 35 \cdot 18\%)$$

With an additional 140 new function points, the combined effective size would equal the effective preexisting function points (116) plus the number of new function points (140), for a total of 256 effective function points. This effective function point measure quantifies the work to be performed and forms a basis for tracking completion.

Integrating Commercial Off-the-Shelf Software

In hardware design, standardized chips are good examples of standard parts. They are well understood components listed in catalogues and they perform very specific functions: they are the physical building blocks of larger designs. Now imagine preexisting, pretested software components that can be inserted right into new software programs.

These pieces of pretested software are like little black boxes. Just like a standard hardware part, commercial off-the-shelf software is meant to be cheaper and more reliable than a home-grown solution. The following are common subcategories for commercial off-the-shelf software:

Table 8.2 Redesign and Reimplementation Breakdown

Redesign Breakdown		
	Formula to compute redesign percentage:	$0.22 \times A + 0.78 \times B + 0.5 \times C + 0.3 \times (1 - (0.22 \times A + 0.78 \times B) \times (3 \times D + E)/4$
Weight	*Redesign Component*	*Definitions*
0.22	Architectural design change (A)	Percentage of preexisting software requiring architectural design change
0.78	Detailed design change (B)	Percentage of preexisting software requiring detailed design change
0.5	Reverse engineering required (C)	Percentage of preexisting software not familiar to developers; requires understanding and/or reverse engineering to achieve modification
0.225	Redocumentation required (D)	Percentage of preexisting software requiring design redocumentation
0.075	Revalidation required (E)	Percentage of preexisting software requiring revalidation with new design
Reimplementation Breakdown		
	Formula to compute re-implementation percentage:	$0.37 \times F + 0.11 \times G + 0.52 \times H$
Weight	*Inputs*	*Definitions*
0.37	Recoding required (F)	Percentage of preexisting software requiring actual code changes
0.11	Code review required (G)	Percentage of preexisting software needing code reviews
0.52	Unit testing required (H)	Percentage of preexisting software requiring unit testing
Retest Breakdown		
	Formula to compute retest percentage:	$0.10 \times J + 0.04 \times K + 0.13 \times L + 0.25 \times M + 0.36 \times N + 0.12 \times P$
0.1	Test plans required (J)	Percentage requiring test plans to be rewritten

Table 8.2 (continued) Redesign and Reimplementation Breakdown

Weight	Inputs	Definitions
0.04	Test procedures required (K)	Percentage requiring test procedures to be identified and written
0.13	Test reports required (L)	Percentage requiring documented test reports
0.25	Test drivers required (M)	Percentage requiring test drivers and simulators to be rewritten
0.36	Integration testing (N)	Percentage requiring integration testing
0.12	Formal testing (P)	Percentage requiring formal demonstration testing

Table 8.3 Example of Reimplementation Calculation

	Formula:	0.37 × F + 0.11 × G + 0.52 × H
	Reimplementation result:	32.82 Percent
Weight	*Inputs*	*Likely Percent*
0.37	Recoding required (F)	30
0.11	Code review required (G)	32
0.52	Unit testing required (H)	35

COTS components — Program parts designed to be included within developed software to provide additional functionality. These parts are designed, developed, tested, documented, and usually maintained by their suppliers. Because COTS components are developed and tested with reuse in mind, they are more generically designed than typical custom software.

COTS applications — Stand-alone applications available for sale to businesses, government, and the general public. For example, word processors such as Microsoft Word or WordPerfect may be considered COTS applications when they are used to fulfill user requirements. COTS applications are clearly tempting because they provide turnkey functionality. However, specific needs may not be perfectly addressed by these general applications.

GOTS (government off-the-shelf) — Similar to COTS, except that the software was developed by or for the government and may not be widely available. GOTS software is generally provided at no cost to government organizations and often to software developers contracting to the government. However, support and updates are not necessarily included or forthcoming. GOTS software has no specific requirements so it may or may not be accompanied by satisfactory documentation. Often it is provided as government-furnished information (GFI) — software without guarantees or warranties —and the developer using it is left with the task of determining how usable it is and what if any of the functionality needed is actually provided.

Examples of COTS software include:

*Stand-alone packages such as word processors and spreadsheets

*Libraries requiring linkage into application code, for example graphics engines, Windows DLLs

*Development environments with runtime modules, for example, Visual Basic™ and Sybase™

*Vendor-supplied device drivers such as printers, displays, and multimedia

*Information retrieval applications such as hypertext and data mining tools

*Operating system utilities such as file operations and memory management

Use of a COTS product implies a certain trust of its vendor. It is often useful, therefore, to learn as much as possible about the vendor — what are its other business obligations and what is its financial condition — before deciding to use its product.

Fundamental Differences between COTS Software and Custom Development

Here is a list of considerations surrounding any evaluation of COTS versus custom development:

*An "infrastructure" may be required to demonstrate and validate the package.

*The COTS package may dictate standards, architecture, and design.

*Because it has a pre-specified design and certain input and output restrictions, COTS may also influence work flow.

*Choosing the wrong COTS package may be more expensive than fixing problems in custom software.

*Resolution of COTS-related issues may be complicated because of the addition of a third party (vendor).

*There may be no source code available and no way to correct a defect.

Items Not Estimated as COTS

When estimating development costs, confusion sometimes arises as to what constitutes COTS and what does not. In general, anything that is purchased for use in a development project is potentially a COTS item. However, software that is used to create software but is not part of a finished product is not COTS; it is a development tool.

Do not cost standard operating systems and development tools as COTS — The impacts of operating systems and development tools are handled explicitly, by experience parameters in various estimating models. These items generally are tools that aid in the development of an end product. Other examples include code scanners, code generators, automatic testers, requirements tools, and configuration management tools.

To consider the use of development tools in an estimate, look at the ratings for such parameters as automated tool use, requirements definition formality, function implementation mechanism, and others that relate to the development environment and processes. Acquisition costs associated with these tools must be included in a complete estimate.

Do not estimate modified COTS in the same manner as non-modified COTS — Just as electronics equipment warranties are no longer valid after a seal is broken, COTS software is no longer COTS after its source code is modified. It may still be estimated as reusable software, but potential COTS advantages are lost:

*The COTS supplier no longer maintains your documentation and source code.

*You no longer know what you are getting because modifications may or may not be consistent with the original software design.

*New updates to the baseline COTS software may not be usable unless modified to suit whatever changes you have made.

*Modified COTS should be handled as incidental or planned reused software (depending on the modifications made); this is still less costly than new development, but not as cost effective as unmodified COTS.

Do not cost incidental and planned reuse software as COTS — Neither of these software types should be treated as COTS software because they are not commercially available and because software code will be worked with.

Weighing Use of COTS

When properly applied, COTS can truly reduce costs, schedules, and development risks. However, several issues related to the use of COTS software must be considered. More than money must often be invested to make a COTS investment work. Listed below are some advantages and disadvantages developers see when evaluating COTS:

Advantages	*Disadvantages*
Quicker time to market	Use involves learning curve; need for integration and further customization
Better reliability	May not meet all user requirements because it is intended for general use
More end user functionality when compared to custom-developed components	Can be difficult to support because source code may not be provided
Support for components across different hardware and environments	Vendors may discontinue support or cease business
Stricter requirements because of its release for general use	

Case Studies: Real-World Experiences with COTS

The following case studies are actual examples from the authors' experiences. COTS usage has been both a blessing a curse.

Case 1: Components Had Critical Defects and Were Modified by Developer

A COTS library was chosen to store and provide instant access to thousands of items of text data associated with inputs. Defects in delivered COTS and its interaction with the developed application made it necessary for the developer to procure source code and debug and/or correct the previously

COTS software. When the COTS vendor moved to a newer version, the developer was stuck with the modified previous version. The result was that the developer had to maintain what was once the COTS portion on its own; the advantages of increased functionality from new versions of the COTS were lost forever. Cost and performance savings over "native" code were still realized, although nowhere near the amounts projected. Fortunately, the COTS vendor made source code available. If source code had not been available (at a price) for this project, the costs would have been even greater, schedule penalties could have been imposed, and development of new solutions would have been required.

Case 2: Powerful (and Defect-Ridden) COTS Component

This project had a graphics display COTS component that was provided as an executable program only, with no source code available. The developer decided to use this COTS package because it provided the best functionality on the market. However, the documentation was poor and the initial releases were defect-ridden. Support was also poor — calls were not always returned. When problems could not be solved, the software requirements were relaxed to make the COTS acceptable. As newer versions of the operating system appeared, however, the COTS software did not always execute properly and vendor support continued to be inconsistent. In addition, the COTS component was available on only one platform. Despite this, the developer assumed that equivalent functionality would be available by the time a second platform was required. The developer also assumed that glue code would be developed to convert calling sequences. No adequate functionality was ever developed for the second platform by the vendor, and the developer was required to develop this functionality from scratch.

Case 3: Application Integrated (Loosely Coupled) without Problems

This application required a simple text editor. There was no requirement for tight data coupling; the only requirement was executing the application. The development team used the operating system's editor. In this case, invoking the stand-alone COTS application was simple, it ran smoothly, and required almost no development time.

Evaluating and Estimating COTS

Imagine that all the work required to integrate a COTS component with the target software is a blob as represented in Figure 8.1. The interface with the COTS functionality appears on the right side. The target software

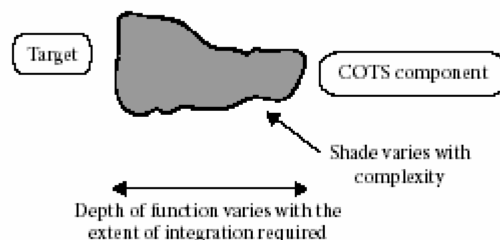

Figure 8.1 Integration of COTS component with target software.

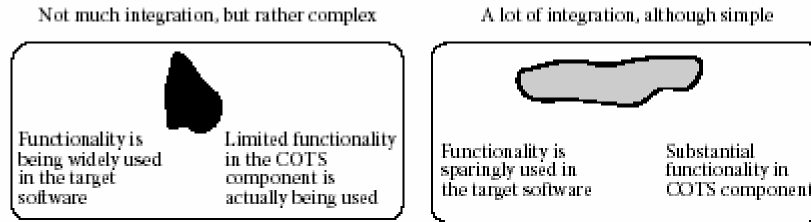

Figure 8.2 Examples of blobs in practice.

interface or resultant functionality impacted is represented on the left side. The area in the middle represents the required work.

The amount of COTS functionality, the impacted target functionality, and integration work together to determine the size of the blob, and thus the job. A lot of integration work means a longer blob and more functionality makes it wider. A fourth factor is complexity, and this is represented by shading. Although complexity is positively correlated with integration work, sometimes a small amount of work will be very complex and vice versa. Figure 8.2 illustrates two examples of blobs in practice. It can be used as a framework to better explain the scope of any COTS integration work.

Three Components of COTS Integration

Figure 8.3 illustrates the three basic components required when estimating the effort involved in integrating COTS software.

Glue code is software that binds COTS software with development software. Glue code can actually serve as an integral part of development software or it can be developed as a separate module inserted between the system being developed and the COTS components. Glue code should

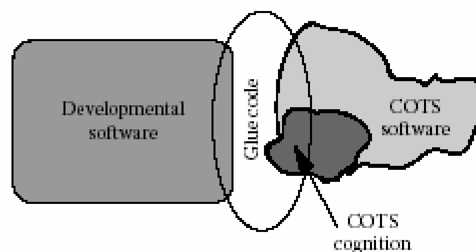

Figure 8.3 Glue code integrating developmental software and COTS software.

normally be modeled as any other coding effort is. It can be sized using either source lines or functions.

Developmental software is generally developed from scratch to meet the stated requirements of a project.

COTS cognition or learning is not a cost-free activity. Even when a developer can directly put a COTS component or application to use, understanding how to use third-party software takes time, effort, and diligence. COTS cognition involves identifying the number of COTS functions that must be learned, used, and tested.

Estimating COTS Integration

This section outlines three different methodologies that can be used to estimate the integration of COTS software: (1) using function points and an estimating model lacking COTS-specific capability; (2) using SEER-SEM cost drivers to estimate COTS; and (3) rules of thumb.

Using Function Points and Estimating Model Lacking COTS-Specific Capability

For the COTS software, prform the following steps:

1. Count the functionality to be learned or used by the developers. In the language of function points, these will usually be internal logical files.

2. Count only the function calls used by the host application as external inquiries.

3. Count any error messages used as external interface files.

4. Count screen, printed, and clipboard outputs individually as external outputs.

5. Count user interactions with library functions as external inputs.

6. In general, count inputs and outputs within the boundary of the host application.

Integration of Stand-Alone COTS Software

What about stand-alone packages that are simply launched from a host application? For example, a mission control system may have an option to launch a word processor. The effort associated with such types of turnkey integration is captured in the host application's development effort. This is appropriate only for COTS packages that are used as-is with no further understanding required. The only required testing would be verification that the host application successfully launches the COTS application with no unwanted side effects.

Stand-Alone COTS Software with Significant Configuration

Some stand-alone packages require significant configuration. For example, a mail system often requires extensive hardware-dependent setup and may involve script and initialization file edits, 4GL code, etc. Configuration should be modeled within a sizing framework, that is, determine the size of each item to be configured. Use the following guidelines for estimating configuration work:

1. Count setup files as internal logical files (function points).

2. If the configuration is performed from scratch, count whatever is created either as SLOC or functions.

3. If configuration templates are provided, count them as preexisting SLOC or functions.

4. For cost drivers when estimating COTS integration without a COTS estimating tool, evaluate the Table 8.4 factors after completing the function point count for the COTS. Even though Table 8.4 expresses the cost drivers in terms of SEER-SEM parameters, they must be evaluated whether using a cost model or simply performing a manual cost study.

Using SEER-SEM Cost Drivers to Estimate COTS

COTS estimation mainly involves sizing the core new developmental code, COTS glue code, and any "cognition" required to integrate and understand the COTS component or application (see Figure 8.3).

Table 8.4 Cost Drivers, Descriptions, and Typical Settings

Potential Cost Driver	*Description*	*Considerations*
Resource and support location	Degree of access (by proximity) to COTS software vendor resources and support	COTS software vendors are usually geographically remote and support levels can vary tremendously; may be a critical issue if software is complex and challenging to use
Host system volatility	Determines difficulty caused by changes to development virtual machine; impacted by how often COTS software is updated	Volatility may be high if developers want to keep up-to-date with vendor releases
Specification level (reliability)	Level of documentation required	Should be adjusted to reflect specifications that will have to be written to support use of COTS product
Test level	Level to which COTS software will be tested	Stringent internal QA levels may require detailed retesting of vendor-supplied software before internal acceptance
Quality assurance (QA) level	QA level to which COTS software was built	Highly vendor- and product-specific
Requirements volatility (change)	Anticipated frequency and scope of change in requirements once baselined (after preliminary design starts)	Rate based on requirements for COTS portion of the project, not overall project
Language type (complexity)	Difficulty of programming languages used in development; parameter is closely related to function implementation mechanism	Rate COTS software based on complexity of its programming support tools and interface

Table 8.4 (continued) Cost Drivers, Descriptions, and Typical Settings

Potential Cost Driver	*Description*	*Considerations*
Memory constraints	Anticipated effort to reduce memory usage	COTS software may use so much memory that conservation measures must be undertaken elsewhere in developed code
Time constraints	Percentage of software that must undergo specific (coding) effort to enhance timing performance	If COTS software is too slow, nothing generally can be done about it; requirement relief or different solution is needed
Real-time code	Amount of software involved in real-time functions driven by a clock external to the software, e.g., gathering data from hardware devices or time-sensitive control of such devices where waiting can alter or lose data	Software that lacks definite and extremely tight time constraints is not real-time code; if COTS has real-time consideration that cannot be met, a different solution is needed
Target system complexity	Complexity of target operating systems, compilers, controllers, and other attached processors	Target system is host development environment; parameter varies with extent of change in that environment
Target system volatility	Determines difficulty caused by changes to virtual machine; may be changes in program editors, compilers or other tools, changes in command languages, or changes in target hardware	Target system is host development environment; parameter varies with extent of change in that environment
Security requirements	Development impacts of security requirements for delivered target system	COTS security levels must usually be accepted as-is; contact vendor for security certification rating

through 7. Table 8.4 illustrates some opportunities and risks, and parameter settings that are associated with COTS software based on SEER-SEM cost drivers. The method of estimating COTS cognition is described below.

A key driver behind COTS estimates is the scope of the integration effort. While traditional size metrics such as lines of code or function points are used to scope traditional work, size is typically not known for COTS integration efforts. In addition, the size of a COTS product is not always correlated with the effort required to integrate it.

For a COTS element, "size" describes the functionality that must be understood by the integrator. This perspective on sizing has been called COTS cognition.

SEER-SEM provides a number of COTS cognition sizing methods: (1) object sizing, (2) feature sizing, and (3) quick sizing which allows an analyst to estimate by drawing analogies. The options for COTS sizing are detailed below:

Object Sizing

If the COTS software is object-oriented, there is no better choice than this metric that was carefully adapted from IFPUG's object-oriented function point specification. Identify the parts of the COTS that will be learned and used, not the entire COTS application.

Feature Sizing

This technique was specially developed for use in the COTS WBS element. Developed from function points, feature sizing allows you to model functions as they appear from the developer's perspective. Feature sizing is broken down into three categories: unique functions, data tables referenced, and data tables configured.

Unique Functions: Defined as the number of unique functions that must be understood to integrate the component, functions may reside in APIs, program libraries, etc. A function may pass data, receive data, or both pass and receive data. Count the number of unique functions used or those that must simply be understood.
Data Tables Referenced: This category represents the number of unique data tables referenced including configuration files, databases, external data structures, etc. A single database having several closely related tables should be referenced once only unless those tables are sufficiently different from one another. These data groupings are referenced only and not changed.
Data Tables Configured: The final category includes the number of data tables that must be configured, changed or created in order to integrate the component. Data tables include configuration files, databases, and external data structures. A single database having several closely related tables should be counted once only, unless those tables are sufficiently different from one another. Count each table being created or configured. If an already existing table is being used only to learn how tables should be created or configured, do not count it.

Quick Sizing

This is a method for approximating size by analogy against common application types. Two categories of COTS software are outlined and discussed here: embedded COTS software and components. Embedded COTS software is integrated directly into the delivered software. Embedded COTS software items are broken down further into those that are adapted and those that are components, that is, directly integrated into the computer program.

Components are intended for reuse; source code often is not available and usually requires no modification such as libraries, object classes, and applications.

What distinguishes the COTS WBS element in SEER-SEM is its list of specialized parameters developed after much research into the critical factors underlying the use of COTS software. This section contains a quick overview of COTS WBS element parameter categories.

Off-the-shelf product characteristics — This category of parameters describes issues faced by users of off-the-shelf components. Integration experiences show that these issues differ from standard development issues, and particularly involve product support and integration complexity. Parameters in this category include component type, component volatility, component application complexity, interface complexity, and product support.

Use parameters relate to the quality of the experiences developers will have in using these components. Many issues surrounding component integration involves learning how to use components and checking their integrity. The use parameters focus on these factors and include component selection completeness, experience with component, learning rate, reverse engineering, component integration and testing, and test level.
Cost parameters are both recurring (e.g., annual licensing fees) and nonrecurring (e.g., one-time purchasing and training costs) associated with the COTS product.

Evaluating COTS-intensive software goes beyond simple recipes. Foremost are cost-performance trade-offs. As an analogy, imagine a prefabricated house. While such a house can be built economically,

Table 8.5 Rough Scope of COTS Implementation[3]

	Scope of COTS (Effort Months)		
Experience	*Limited*	*Moderate*	*Complex*
Limited	2.3 to 22	19 to 99	143 to 787
Functional	1.5 to 16	13 to 72	96 to 582
Fully proficient	1.4 to 13	11 to 58	84 to 464

the placements of doors and windows are prescribed and customizing such a house can be difficult. The same principle applies to COTS software, where savings of time and effort may be traded for flexibility. Your task is to evaluate the additional costs such losses in flexibility may entail.

Rules of Thumb for COTS Integration

Table 8.5 provides general guidance on COTS integration. Use this table to perform sanity checks on integration efforts from the time necessary to understand the COTS component (cognition) through the time necessary to complete integration (completion of glue code and other necessary con-figurations). COTS projects vary widely in the level of required integration, the type of integration carried out, the type of product, ease of use, and many other factors. Because of this, this table must be evaluated skeptically; use it only to understand probable ranges for COTS efforts.

Experience with COTS Product

This factor describes the developers' previous experience in integrating developed software with this COTS product:

Limited: nearly no experience with product
Functional: one or two limited instances of dealing with product
Fully proficient: fully knowledgeable and practiced in integration issues with product

Scope of COTS

Scope is the combined functionality and complexity of the COTS product:

Limited: typical desktop-level software products such as a PCbased accounting system or operating system changeover
Moderate: medium-sized company tools such as a human resources system, shop floor automation, shipping, and operating system changeovers
Complex: mainframe-class products such as SAP R/3, a broadly deployed reservations system, or a system with complex real-world interactions that do not become fully apparent except over time

Table 8.5 does not consider outliers. For example, consider a simple fire-and-forget COTS integration of launching a dedicated word processor. This application, called from a host application, may have a great deal of functionality but could be quite simple to integrate. While experience with an application may be limited, integration can be a trivial matter. COTS integration outliers are more pronounced on the downside of the above estimates.

Evaluation and Selection of COTS Products

Although evaluation and selection (E&S) is often a time-consuming process in its own right, no strong methodology that can be used to estimate this task is available. Look to previous E&S efforts to get a sense for how long a particular effort should take. However, E&S cost is more often a function of budget, continuing until either a best selection is found or until funds are exhausted. Table 8.6 provides a starting checklist for selecting and evaluating COTS products.

COTS Risks

Because COTS software is designed to address common needs, the specifications for this software often sound very appealing. However, a number of common assumptions should be questioned:

***Assumption** — A COTS package is relatively bug-free.
Reality — Although the marketplace tries to ensure that bugs are discovered and fixed promptly, newer, less tested versions cause defects to reappear. Shop for a mature COTS package (or version).

***Assumption** — System integrators know the functionality and interface nuances of the COTS packages that they propose to use.
Reality — Manuals do not always tell the whole story, especially with programming components. Use of almost any COTS involves a learning curve. Look for previous experience with the specific COTS package.

Table 8.6 Checklist for COTS Evaluation and Selection

COTS Characteristic	*Estimation Impact*
Does developer's organization already have experience with this COTS software?	Yes, reduces cost
Is COTS software vendor an established company or a garage shop operation?	Established firms reduce risk
Is source code available?	Yes, reduces risk if vendor is shaky
Does developer plan to use modified COTS?	Yes, means higher costs of rework
What are licensing terms for COTS?	Difficult licensing agreements can cause delay in delivery of product to development organization; licensing fees can significantly impact life cycle costs of software
What are vendor's commitments to upgrades?	If vendor has no commitment to future upgrades, COTS product can require significant modifications by development team; in a worst case, a new product may need to be purchased or functionality may need to be developed
What are software developer's commitments to upgrades (i.e., is delivered product required to be delivered with most recent COTS version)?	If version 1.0 works well in software development, but you are required to deliver the latest version and it does not work the same as version 1.0, costs increase; if version 1.0 requires development team to design work-arounds when the new version comes out, these may no longer be valid
Quality and reliability of COTS product?	Quality and reliability of COTS product will have a direct correlation to the number of modifications and work-arounds required; unreliable COTS can cause entire application to be unreliable

***Assumption** —Glue code is very easy to write. Therefore, only a minimum amount of time is required to design and implement it.
Reality — Because glue code interfaces independent sets of code, it can be extremely complex.

***Assumption** — COTS software works and no special testing beyond integration testing is needed.
Reality — The golden rule behind COTS software is "trust but verify."

***Assumption** — User requirements will be met by the finished system.
Reality — In general, COTS packages need to be extended to meet 100 percent of user requirements.

***Assumption** — COTS is mass-produced and priced at dramatic economies of scale.
Reality — Keep in mind that although COTS software may be sold to many people, developers must still make decent returns on their investments.

Risk Reduction

Maximize the use of COTS components that:

*Perform relatively well defined functions.
*Are mature, replaceable, and have equivalents from alternate, competitive sources. This means studying the market since products are not generally standardized.
*Have well defined and predictable inputs and outputs. This requires looking at product architecture in some detail.

Minimize the use of COTS components that:

*Combine many functions in an integrated solution, all of which must be used or purchased and maintained. Ask whether you can use a portion of the COTS product or package.

*Do not have well defined internal and external interfaces.

Incorporate the other rules of estimation. All the other cautions related to estimation of software projects are applicable to projects incorporating COTS software:

*Experience levels with languages, tools, and practices are just as important as the languages, tools, and practices being used.

*Simply having tools available does not mean they will be used.

*The complexity of an application is related to how quickly you can add people to the project.

*There is no such thing as a free lunch! COTS integration is never free.

Risks Associated with Reuse and COTS

While reuse or COTS software can provide the potential to significantly reduce cost, schedule, or quality exposure of a project, opting for reused or COTS software by no means ensures success. Many risks can arise. If not managed and successfully mitigated, they can develop into problems and negate any savings projected. The risks generally fall into three categories:

1. The process used to select the components or criteria used in trade-offs resulting in the selection
2. A need to modify, extend, or upgrade reused or COTS components to support operational or application requirements
3. The need to maintain or sustain the reused or COTS components during periods of operational support

Table 8.7 identifies certain issues that should be considered when selecting COTS or reused components and the risks that may result. Ignoring these issues and failing to manage the risks can quickly preclude meeting the cost and schedule savings.

Summary

The primary difference between reuse and COTS is the origin of the software. COTS is generally purchased. Incidental reuse has been occurring ever since software was invented and has been a hoped for silver bullet. Reuse is not free nor is it generally inexpensive. Planned reuse can reduce cost but cost more to develop.

COTS product integration has expanded dramatically in recent years as an important strategy for achieving cost-effective software systems. Success of this reuse type has been based on the increasing quality of COTS software products and the growth of technologies supporting the integration of architectural styles such as middleware. Financial issues and concerns as well as improved returns on

investments resulting from better products have increased the pressure to achieve more with development dollars in less time, thus driving system architects to accept and use COTS. These cost factors constitute the primary reason that many organizations are integrating existing systems with new systems and developing software applications that make heavy use of COTS solutions. Only through methodical software reuse, systematic evaluation techniques, and improved understanding of integration can development dollars, time, and effort be more effectively realized with a COTS or reuse development strategy.

Table 8.7 COTS Issues and Risks

Reuse Issue Area	*Risk*
Component Selection Issues and Risks	
Reuse code or COTS not adequately analyzed at program inception before start of architecture design	Component does not integrate with architecture
Inadequate quantified selection criteria and acceptability thresholds	Components inconsistent with application requirements and must be modified or replaced
Fit of functionality to current application not adequately evaluated prior to selection	Component functionality must be modified or enhanced
Adequate cost analysis or trade-off not conducted to identify specific possible cost savings by minimizing code and design modifications and cost of integration related to selection and use of reused or COTS component	Projected savings may prove unreasonable
Compliance of external and system software interfaces with external engineering interface, application program interface, and data interoperability standards not confirmed prior to selection	Component fails to interoperate with other software components; must be modified
Component Modification Issues and Risks	
Interfaces inconsistent with software architecture operating systems and middleware	Component fails to interoperate with other software components; must be modified
Reuse architecture to which designed is inconsistent with project quality standards or software architecture	Components inconsistent with application requirements; must be modified or replaced
Component will not perform adequately under stress conditions	Components inconsistent with application requirements; must be modified or replaced
Range of values for input variables inconsistent with those required by software architecture	Components inconsistent with application requirements; must be modified or replaced

Table 8.7 (continued) **COTS Issues and Risks**

Reuse Issue Area	*Risk*
Security characteristics of component inconsistent with those required for application	Components are inconsistent with application requirements and must be modified or replaced.
Documentation or support elements related to component may be of insufficient quality to be used, may not be current, or may not meet project standards	Documentation or support elements may require modification or replacement; costs may increase for reverse engineering effort required to understand poorly documented COTS
COTS or reuse code may not be a fit with functional requirements it is designed to satisfy	May require design modifications and recoding that will increase cost of integration
No trade study conducted to evaluate reuse or COTS code before or after architecture design	Components inconsistent with architecture requirements; must be modified or replaced
Architecture was designed before reuse or COTS code modules were selected, making integration into architecture with least modification unlikely	Architecture inconsistent with reuse or COTS requirements; component or architecture must be modified or replaced to use it
Need for costly development of "wrappers" to translate reuse or COTS software external interfaces	Unexpected costs or schedule may result from need to develop and qualify wrappers essential to integrate components into software architecture
Sustainment Issues and Risks	
Proprietary features of COTS product and positive and negative impacts of all features not identified early	Maintenance of product is impacted or precluded due to proprietary nature of COTS or reused components
Processes used to develop or update COTS or reuse product not sufficiently rigorous to assure that excessive defects do not remain in product	Product may not perform well enough to support needs of application
Applications operating on client–server network must analyze ease of distributing COTS product and its output data on network	Licenses for COTS components may restrict use to limited number of machines

Table 8.7 (continued) COTS Issues and Risks

Reuse Issue Area	*Risk*
Analysis of sustainment costs for each candidate module not conducted before selection of COTS or reuse software	The cost of sustainment may exceed initial projections
Cost of upgrading to new versions including incorporating previously made modifications and additions not identified before selecting product	Sustainment costs may exceed projected costs and schedule
Costs of licenses not adequately determined prior to selecting COTS component	COTS licensing costs may exceed projections
Costs of replacing product due to proprietary features may not have been calculated or were calculated improperly	Costs for COTS components requiring replacement due to proprietary factors may be excessive
Process used for sustainment of COTS or reused components may not be sufficiently robust to assure long-term integrity of component	As product modifications are applied, integrity of component may degrade
Inadequate funding to keep reuse or COTS current with evolving hardware and system software	It may not be possible to keep components current with technology or application requirements due to resource limitations
Configuration management and control of COTS and reuse products by sustainment organization may prove inadequate to control product releases and changes	COTS and reused component baselines may be lost and unauthorized changes may be applied
Product help desk inadequate to assist users of COTS or reuse components who require assistance	COTS or reuse users may not be able to resolve issues with components
COTS vendor may go out of business	If source code was delivered, development organization may have to take over maintenance of package or new COTS package may need to be purchased, integrated, and deployed

Every day, reusing software or applying a piece of COTS software to satisfy an operational or user need becomes easier as more and more vendors offer more and better software products for a dizzying variety of applications.[4] As Northrop Grumman found, COTS effort can be estimated successfully: “The overall effort and schedule predicted were within two
percent of the program actuals.”[5]

Endnotes

1. Reifer, Donald J. *Practical Software Reuse*. New York: John Wiley & Sons, 1997.
2. Galorath Incorporated. *SEER-SEM User Manual*. El Segundo: Galorath Incorporated, 2004.
3. Galorath Incorporated. *OSD Software Estimation Guidebook*. El Segundo: Galorath Incorporated, 1997.
4. Brooks, Frederick P., Jr. “No Silver Bullet: Essence and Accidents of Software Engineering.” *Software*

Magazine, 1995.
5. Bradford, Kathy and Lori Vaughan. *Improve Commercial-off-the-Shelf (COTS) Integration Estimates*. Redondo Beach: Northrup Grumman Mission Systems, 2004.

For more information, contact Galorath Incorporated, 100 N. Sepulveda Blvd., Suite 1801, El Segundo, California 90245. Phone: 310-414-3222, Fax: 310-414-3220, E-mail: info@galorath.com, Web site: www.galorath.com

Originally printed in *Software Sizing, Estimation, and Risk Management.* Reprinted with permission of Auerbach Publications, Taylor & Francis Group.

###