Enterprise Engineering: U.S. Air Force Combat Support Integration

Eric Z. Maass
Lockheed Martin Mission Systems

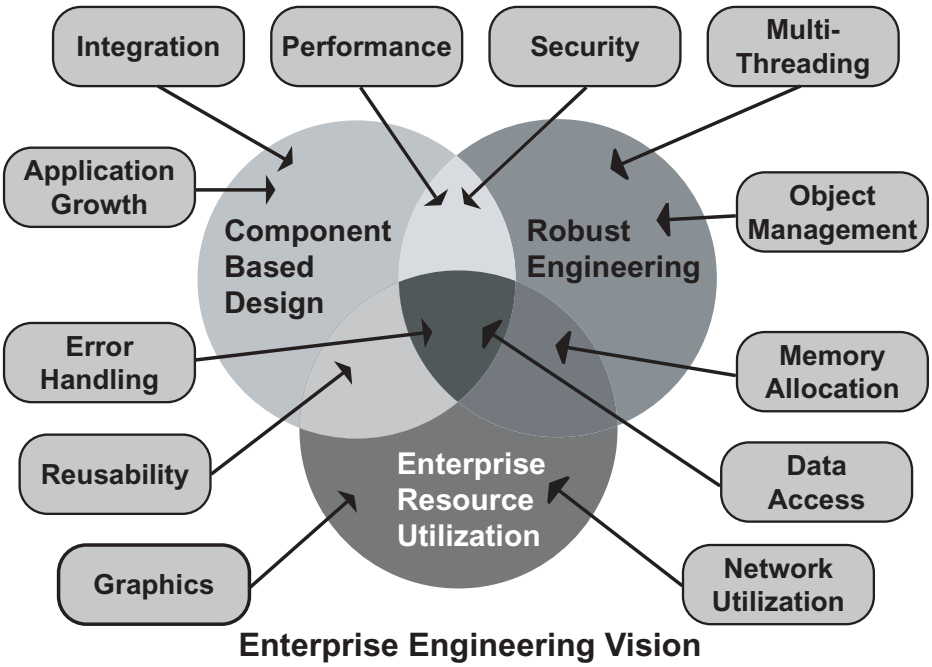
Enterprise engineering is quickly becoming a hot term used to describe the future movement of software engineering. This movement, for both management and development organizations, can be daunting. Application developers will find though that this model is useful in building a common enterprise from disparate organizations. This article covers fundamental considerations for developing to an enterprise engineering vision and discusses basic techniques of enterprise application development as lessons learned on the U.S. Air Force's Global Combat Support System enterprise.

The term enterprise engineering describes a large gamut of engineering practices and processes that enable an organization to design, develop, stand up, and maintain an enterprise-computing environment. Facets of enterprise engineering might range anywhere from models such as time-based competition, continuous improvement, and business process reengineering to the real *meat and potatoes* of enterprise application design, deployment, and integration (enterprise application integration) [1]. However, from a technical engineering perspective, how does a large organization tackle the daunting task of integrating numerous enterprise-class applications while maintaining a single engineering vision? Furthermore, how do multiple sub-organizational structures work to develop first-class enterprise applications for a single, common infrastructure, enterprise environment?

The first realization that must be addressed is that enterprise engineering is the future of the software engineering world. Software, whether it is designed for a home computer, large computing environment, or mobile computing device, is becoming increasingly powerful due to its ability to interact, or integrate, with other systems and their respective software. For instance, after placing an order to purchase an item on an Internet-based store, typically the buyer would receive an e-mail with the status of the order. To make this happen, it could mean integration of several systems and their software: a credit card processing system to verify the credit card used, a warehouse system to see if the item is in stock, a third-party delivery system to prepare delivery of the item, etc.

In the past, these systems were either not integrated or were integrated in some proprietary manner, which made them very costly and not very flexible. Today, the idea of enterprise engineering is sweeping the software engineering world with technologies such as Java 2 Enterprise Edition (J2EE), Web services, and enterprise application integration tools. Hence, we have the introduction of enterprise engineering – the vision of engineering software and systems that integrate across various systems, use open standards, and bridge the gaps between legacy, stovepipe system mentalities (see Figure 1).

Figure 1: *Enterprise Engineering Vision*



In the case of the Air Force, building an integrated combat support enterprise is key to providing real-time, accurate, integrated information to the warfighter. Allowing stand-alone combat support systems to integrate in an efficient, robust, secure manner vastly improves the value of information available to those making combat decisions. For example, being able to integrate combat support systems could provide a single user with the capability of seeing not only what planes are available for a mission, but also their mission readiness, available personnel, financial data relating to making that mission happen, and so forth. Previously, this data might only be available through many separate systems that might have conflicting information due to the fact that they were not integrated and shared the same real-time information.

Now that the end goal is clear, the question becomes how to get there. A key to success in such an enterprise engineering environment is to become a heavily integrated organization built upon common services and open standards, and employing rigorous application development standards in the software engineering processes of the applications joining the enterprise. Of course, there are multitudes of accompanying business models that will wrapper and compliment these engineering practices.

The U.S. Air Force is currently creating such an enterprise by building an integrated environment – one enterprise – that is the platform of integration for the Air Force's vast combat computing systems. The enterprise, known as Global Combat Support System-Air Force (GCSS-AF), draws upon currently operational mission

applications in the combat support arena to modernize applications in accordance with a new integrated enterprise engineering vision. In doing so, from a pure engineering perspective, there are many obstacles to contend with – especially when coordinating the efforts of multiple software development organizations to integrate under one enterprise.

One of the first technical obstacles is enabling developers to understand the vision! Within that vision are the technical details regarding the enterprise architecture, its shared and common services, components and methods of integration, compliance standards, and much more. This article looks more closely at a subset of these aforementioned aspects of an enterprise, some commonly trusted best practices when developing to such an enterprise, and how these issues relate directly to the Air Force's GCSS-AF initiative. The enterprise engineering vision, being the super-set of the technical aspects that will be discussed within this article, can be viewed in Figure 1.

Building an Enterprise Engineering Vision

Building an enterprise engineering vision is one of the most important milestones of successfully constructing an integrated enterprise environment on such a grand scale as the U.S. Air Force combat support computing structure. The enterprise engineering vision is fundamental to ensuring that disparate software development organizations understand their role in the enterprise.

The GCSS-AF program delivers the technical components of its enterprise engineering vision through a platform – an infrastructure for development – known as the Integration Framework (IF). The IF, a conglomerate of commercial off-the-shelf products in a n-tier, Web-based, J2EE-enabled architecture, provides a set of common services and components for applications that join the enterprise. The framework provides a *living space* for application integration that enforces standards while providing a way for applications to join the enterprise and reduce the cost of software development by avoiding reintroduction of common services (such as security, messaging, and data warehousing).

The diagram depicted in Figure 2 demonstrates a typical set of common services offered by the IF on GCSS-AF in a four-tier enterprise. These tiers represent a set of common services that are available to application developers as part of their design for joining the enterprise.

Figure 2: *The Integration Framework on GCSS-AF*

The GCSS-AF environment, designed to host a wide variety of disparate applications in a manner that is conducive to forming a single, heavily integrated enterprise, is also faced with an interesting engineering case: How does an organization collectively ensure that such a conglomerated environment operates efficiently and successfully?

Modern Problems: A Return to Basics

The idea of enterprise engineering may be new to many development organizations, but the fundamentals of making it *work* are largely based on a model of software engineering practices that far outdate the modern concepts of enterprise engineering and most of the technologies that may be present in that environment.

Each application is part of a phased approach at reaching a vision of an integrated enterprise. This means that enforcing the use of the technologies present in the enterprise will be key to following the enterprise engineering vision; on that same note, flexibility in the technology set present in the enterprise is also important. However, perhaps even more important is ensuring that these services and technologies are being implemented correctly. This brings us back to the basics!

Java may be a relatively new programming language, but it shares much in common with its ancestors. This is relatively true for most modern technologies – they display tendencies and traits from their ancestors that are important to note because they may largely aid in successfully implementing them in similar or new ways than was done previously.

Let us quickly review some of the basics of good software engineering practices and see how these might be applied to modern-day enterprise engineering in an environment such as GCSS-AF.

Component-Based Design

Every modern developer has heard the term component-based design. The real question is how many modern developers fully extend its theory into their practices? Furthermore, how many modern enterprise developers and architects consider component-based design theories an integral part of their work? The answer is always quite simple – not enough! [3]

When we speak in terms of Java enterprise development, component-based design should be one of our first thoughts. Java, being a multiplatform-compatible programming language, employs technologies that tend to slow code execution (in comparison to older, single-platform languages) which, at the same time, makes the language flexible. Unlike languages in the past, Java therefore requires additional attention to component-based design so that the applications created in these environments can perform on par with older, quicker languages.

For example, take the following pseudo code design of procedures in a Pascal-like application (see Figure 3, page 18) that might be designed as follows:

```
Function calcTotal (var userTotal : integer);

Begin
        grandTotal := userTotal + (userTotal * salesTaxPer);
        writeln 'Your total is: ' + grandTotal;
end; {calcTotal}

Function on MailingList (var mail : boolean);

Begin
        {... lookup if customer is on mailing list ...}
        {... access private account data ...}
end; {onMailingList}

Procedure initProgram;

Begin
        writeln 'XYZ Company System';
        if user.onmailinglist = true then
                initShoppingChart;
end;
```

Figure 3: *Pseudo Design Example: Pascal-Like Application*

This pseudo code snippet demonstrates a function and procedure call in a Pascal-like application. The function, *CalcTotal*, is used to calculate a user's shopping total by adding the necessary sales tax and perhaps other related calculations. It then displays the grand total.

The second, a function called *onMailingList*, is used to determine whether a customer is on the company's mailing list. The procedure does some set of operations (most likely accessing some kind of database) and would return a true or false Boolean value reflecting the customer's mailing list status. Next, we see that the function accesses some private account data related to the customer previously looked up for membership on the mailing list.

Finally, we have a procedure that initializes the application. It most likely would perform multiple operations to initialize the application such as displaying the company name. In addition, this procedure also determines whether or not to initialize the user's *shopping cart*.

Agreeably enough, the above example is not a great example of component-based design, but in the case of a stand-alone, single-user Pascal-like application, the above would probably not be a terrible setback for application performance. In an enterprise engineering environment like GCSS-AF, however, this example would be a major contributing factor to deterioration of system resources, application performance, and overall enterprise engineering vision for high-performance first-class integration because the key fundamentals of sound, component-based design have been overlooked.

In an enterprise with a vision of full application integration, certain items are of key concern [4]. These are explained in the following sections.

Performance

Component-based design is *necessary* for obtaining higher levels of enterprise application performance. Separating out code that is unnecessary for execution will help mitigate the problems seen in slower languages such as Java.

As seen in the example, had the programmer separated out the arbitrarily placed initialization of the user's shopping cart and moved this functionality to a more appropriate part of the code, the application would initialize quicker. For instance, the programmer might have decided to determine whether to initialize the shopping cart only after the user elected to *go shopping* rather than just assuming the user was ready to shop.

"Today, the idea of enterprise engineering is sweeping the software engineering world with technologies such as Java 2 Enterprise Edition (J2EE), Web services, and enterprise application integration tools. Hence, we have the introduction of enterprise engineering."

Reusability

Component-based design is also *necessary* for obtaining code reusability. Although this may not be particularly necessary in all initial instances, having code in a reusable format allows for quick transitioning to higher levels of integration such as allowing an application to offer parts of its functionality as a Web service.

In the earlier example, the programmer should have broken out the function CalcTotal into two separate functions: one for calculating the total, and another for displaying the results. This micro-managed modular approach would (on a grander scale) help the programmer quickly share functionality and reusability of that functionality in the future.

In the world of enterprise engineering, this becomes especially important and is an intricate part of the J2EE model. The earlier example would therefore translate to having all Hyper-Text Markup Language generated in Java Server Page code for displaying the total while the calculation itself would take place, perhaps, in a servlet or Enterprise Java Bean.

Security

Security is often overlooked and becomes an afterthought in application design. Component-based design is also necessary for ensuring that the proper security is in place for an enterprise application.

In the previous example, we see no indications of security measurements. However, we might assume that in an enterprise application (especially one residing in the GCSS-AF environment), sensitive data will need to be filtered, restricted, and monitored. Therefore, implementing method-level security is, typically, necessary in all applications.

In the function onMailingList, private customer data is accessed after the user has been looked up on the mailing list. If this functionality is not necessary for determining whether a user is on the mailing list or not, it should be broken out into a separate, secured procedure. Even if the private customer data is not made available to the application user in the procedure onMailingList, an application error, for example, could cause unexpected exposure of the data or privileged functionality.

Learning to Live Inside the Box

Generally speaking, we would like application developers to *think outside the box* while still realistically considering that their application must *live inside the box*. Enterprise engineering is ultimately the balance between the two.

When applications decide to join the GCSS-AF enterprise, multiple considerations need to be made in order to account for the application's capacity resources, performance requirements, compatibility with other GCSS-AF applications, and a multitude of other facets that may impact the application's ability to reside onboard the program. While a good number of these considerations may be business-process related, another good number is purely engineering issues that must be addressed during an application's design phase.

Again, we are taking some steps back

to the basics, but emphasis on these techniques and viewpoints may, in the end, determine an application's success on GCSS-AF or any other enterprise environment [4].

Memory Allocation

Allocating too little or too much memory is often not detrimental to a smaller standalone application; however, when in a J2EE environment, memory handling becomes increasingly important as both the application and enterprise weigh in [4].

Common memory allocation problems are as simple as using efficiency when dealing with data types in an application. For instance, allocating a 30-character array for a 10-digit code may waste 40 bytes of memory per user executing that code snippet. With potentially thousands of users executing that same code simultaneously, one simple programming error due to negligence could lead to a significant amount of memory waste. An application containing many of these same mistakes in conjunction with other forms of memory allocation errors could easily bring down the application and other applications in the enterprise that are either dependent upon shared resources or services provided by this application.

Multi-Threading

Multi-threading is an application architecture design point intended primarily to allow an application to perform multiple tasks at once in a safe, highly efficient manner.

Multi-threading an enterprise application is considerably important. Most J2EE operations performed in a Web-based application environment should be designed as asynchronous, multi-threaded calls. Depending on synchronous operations can drastically impede an enterprise's performance [2].

Data Access

Remember that your *common services* are generally shared resources that are accessed by various other applications. This includes your data resources. It is important to keep in mind that database connections should be pooled, take advantage of extensible architecture (XA)-compliant database drivers, and be used as efficiently as possible.

A few examples of this may include the following: accessing a database connection, Transmission Control Protocol/Internet Protocol connection, File Transfer Protocol connection, or other connection type to an external resource only when necessary [2]. As well, when connected to the data resource, make sure data are created, read, updated, and deleted in the most efficient natures. For instance, search a table based upon an index – avoid scanning the entire table. Other considerations may include terminating connections when not in use, simplifying data storage schemes (size and complexity of records), transferring only necessary parts of a record, and using the smallest, most efficient data types in records (i.e., the abbreviation AL instead of Alabama).

Error Handling

A single application's stability can potentially impact the stability of the enterprise. For example, if an application's faulty code continually tries to poll an enterprise resource every second with a large query due to a failure in the application, the result would be troublesome for the enterprise resource and every other application depending upon the availability of that resource [4]. Many such problems can be avoided with the use of rigorous error handling and capturing [3]. Employing tight regulations for error handling when performing operations that may impact the enterprise is extremely important and must be designed into the application's functionality.

"Building an enterprise engineering vision is one of the most important milestones of successfully constructing an integrated enterprise environment on such a grand scale as the U.S. Air Force combat support computing structure."

Clean Up

Garbage collection is Java's native way of conserving resources [2]. However, performing such maintenance as garbage collection is the obligation of design. Unused objects should be discarded to conserve memory resources; database records that are no longer necessary should be deleted; databases that are frequently changed should be *reorganized* for performance. Keeping an application's workspace and footprint small, tight, efficient, and clean will help the entire enterprise be successful!

Help Keep the Roads Clear

Network congestion is one of the top causes of poor Quality of Service (QoS). In an enterprise, QoS is a shared responsibility that starts with the QoS initiatives of each application residing in the environment.

A first-class enterprise application is just as concerned with QoS as it is with functionality and robustness of code. QoS typically includes everything from usability of user interfaces to responsiveness of requests to the application; however, one major aspect of QoS that will hit every application hard will be network congestion.

Typically, as an enterprise application residing in a conglomerated environment, the standards or availability of network resources can be somewhat questionable. Therefore, cleverly designing your application to avoid potential QoS problems is typically a wise decision. The following sections are a few examples of design aspects that would be beneficial to an enterprise application's QoS.

Graphics

Graphics are notorious for causing poor QoS and are generally unnecessary for most applications. If your application must employ graphics, some general guidelines should be followed:

- Use black and white graphics if possible.
- If color graphics are necessary, use the lowest bit-depth possible (8-bit, 256 colors) to reduce image size.
- Use the highest compression available on file formats. Using JPG or GIF formats above BMP formats is such an example.
- Keep the image dimensions as small as possible while still keeping effective its business purpose.
- Allow users the option of viewing graphics instead of displaying them by default.
- Display a minimum number of graphics per page.

Packet Trips

Avoid using multiple round trips to achieve what could be done in a more efficient, perhaps larger, transmission. This also includes reducing the amount of data that is retransmitted or checked for

integrity.

A simple example in a Web-based application would be static data retransmission. Employing frames in an application can maintain static data on the user's screen while allowing only the necessary frame to update with a request.

Data Transmission

Send and collect only necessary data. Avoid sending or collecting extraneous data that would utilize enterprise resources without gainful purpose. Some examples of this may include the following:

- Large cookies containing consequential or infrequently used data.
- Dynamic Web objects that require a high frequency of refreshing (i.e., syndicated news).
- Related but unrequested data. Giving the user options to view this data conserves resources rather than sending this data by default.

Keys to Success

From a developer's standpoint, enterprise engineering in a conglomerated enterprise is not an easy task. However, as we have reviewed here, sometimes taking a step backwards and understanding the basics is the key to providing a strong foundation for an enterprise application. There will inevitably be a gamut of hurdles to jump over regarding application-specific engineering, business process models, and so forth; but, attacking the basic engineering from these guidelines and building these guidelines into the application's model will also inevitably aid the success of the application and the enterprise as a whole.

In review, the points to remember include the following:

- Design to the enterprise engineering vision. Using (correctly) the resources available from the enterprise will avoid unnecessary development efforts and help an application integrate into the rest of the environment.
- Practice healthy, component-based design techniques. Component-based design has extremely important benefits in the enterprise, including performance, code reusability, and security.
- Pay rigorous attention to detail. Rigorous attention to programming details will affect the success of an enterprise application to a much greater degree over a stand-alone application.
- Conserve network resources. Enterprise applications must share network resources. Paying attention to these details during design time will allow all applications in the enterprise to increase their QoS.

Enterprise engineering is the future of large-scale organizational computing. Understanding how to develop well performing, integrated applications for such an environment is the beginning step for a successful experience.◆

References

1. Enterprise Engineering: An Information Systems Perspective. 27 Feb. 2003 <www.eil.utoronto.ca/ papers/ mikePapers/eeg16.html>.
2. Farley, Jim, William Crawford, and David Flanagan. Java Enterprise in a Nutshell. Sebastopol, CA: O'Reilly and Associates, 2002.
3. Joines, Stacy, Ruth Willenborg, and Ken Hygh. Performance Analysis for Java Web Sites. Boston, MA: Pearson Education, Inc., 2003.
4. Maass, Eric. "Application Performance for GCSS-AF." GCSS-AF Guide to Developing With the Integration Framework. June 2002.

About the Author

Eric Z. Maass is a software systems engineer for Lockheed Martin Mission Systems in Owego, N.Y., a Capability Maturity Model Level 5 organization. As a member of the software integration and development team on the Air Force's Global Combat Support System (GCSS-AF) program, Maass' primary responsibilities include leading and supporting architecture, development, and integration of the enterprise's security services, leading systems performance optimization research for the GCSS-AF production enterprise, and providing engineering support for application integration into the GCSS-AF program. Maass was recognized in 2002 as a GCSS-AF Top Contributor. He is a graduate of Syracuse University.

Lockheed Martin Mission Systems
1801 State Route 17C
MD 0605
Owego, NY 13827
Phone: (607) 751-2293
Fax: (607) 751-2538
E-mail: eric.maass@lmco.com