An Architectural Approach to Building Systems from COTS Software Components

Vigder, Mark; Dean, John

An Architectural Approach to Building Systems from COTS Software Components

Dr. Mark R. Vigder
John Dean
National Research Council
{mark.vigder | john.dean}@nrc.ca

Abstract

As software systems become increasingly complex to build developers are turning more and more to integrating pre-built components from third party developers into their systems. This use of Commercial Off-The-Shelf (COTS) software components in system construction presents new challenges to system architects and designers. This paper is an experience report that describes issues raised when integrating COTS components, outlines strategies for integration, and presents some informal rules we have developed that ease the development and maintenance of such systems.

1. Introduction

Modern software systems are becoming increasingly expensive to build and maintain and users are becoming more sophisticated in terms of the capability they expect. To build such systems, developers must use a large number of standards, protocols, technologies, and tool kits, each one of which is complex and involves a steep learning curve. Development organizations have met this challenge by using off-the-shelf software components that have been developed outside their organization and which provide much of the functionality and capability required, rather than building their own components.

Components that are bought from a third-party vendor and integrated into a system are defined as *Commercial Off-The-Shelf* (COTS) software components. Building a system from a set of COTS components introduces a different set of problems than building a system from scratch or building a system by re-using components that have been previously constructed internally in the development organization [5,12]. Many of these problems are introduced because of the nature of COTS components: they are truly black-box and the developers have no method of looking inside the box; developers have little or no influence over the maintenance and evolution of the components; and the behaviour of the component may be inadequately specified to understand its behaviour in a multi-component system. Often the COTS component is meant to run as a standalone application and has no mechanism for interacting with other programs.

In order to address these problems we have been experimenting with building systems by integrating COTS components. Among the objectives of these experiments is to look at: technologies that support component integration such as CORBA or ActiveX; languages that are useful for gluing components together [2,8,14,15,17]; and system architectures for using COTS components [3,6,10,11]. We are looking at these problems from the perspective of the integrator using COTS components rather then from the perspective of the builder of the COTS components. This paper is an experience report that describes issues raised when integrating COTS components, outlines a software architecture for integration, and presents some informal rules we have

developed that ease the development and maintenance of such systems.

2. Building Systems from COTS Software

COTS software is a software component that a developer acquires from a third-party and integrates into their system. The COTS component supplier has the component ready-built and is supplying and supporting the identical component for numerous customers.

Developers have been using COTS components for many years [4,6,7,9...]. Traditional COTS components include operating systems, databases, and procedural libraries. Newer examples of COTS components include: complete stand-alone software systems that are being extended or integrated with other applications; components built using the emerging component standards such as ActiveX, JavaBeans, or CORBA; and application frameworks that a developer can tailor using inheritance or plug-ins.

The characteristics of COTS software that makes the software development process different from using custom-built components are:

- Developers do not have access to source code. Because they do not have access to source code developers cannot modify the code to change the functionality of the component (perhaps a good thing!). It also means that analysis, instrumentation, and testing of the component must be done in a totally black box manner.
- The component user has little or no control over the evolution of the system. The system developer who is integrating COTS components is simply one of many customers to the COTS vendor. The developer does not control, and may have minimal influence, over how the component evolves. The functionality added to each update of the component may not be as required by the developer, it may not be ported to the platforms the developer requires, it may interfere with the operation of some required functionality, or it may interact in some unexpected way with other components.
- Complete and correct behavioural specifications are not available. The specifications provided for COTS components are not always correct nor complete. Even if the COTS vendor provides a functional description, this does not always satisfy the needs of the integrator who may need to know more detailed behavioural specifications and resource requirements of the COTS component. Integrators may use COTS components in ways not foreseen by the COTS vendor.
- Sets of COTS components may be mismatched. The mismatch between components can arise for many reasons [6] such as the data model, functional mismatch, resource clash, or process model used. Sometimes the mismatches are not found until quite late in the development process.
- Many COTS components are designed as standalone applications and may not easily interact with other COTS or developmental software.

A COTS based development is fundamentally a problem of integrating black-box components rather than building components. This integration process is not easy [6,12]. It is error prone, requires significant amount of coding, and is difficult to test and debug. In addition, many COTS components have a high level of volatility. Commercial components are often subject to frequent upgrades. These upgrades may not have the added functionality/bug fixes desired by the integrator. Critical functionality which existed in a previous version may have been removed in a subsequent upgrade. In some cases the integrator may wish to substitute similar components from different vendors in new releases of the system.

In order to be able to deal with the construction problems there are a number of properties that are desirable for an

architecture that integrates COTS components.

- Plug-and-play of components. The architecture of the system must allow the substitution of components. Component substitution can involve substituting one version of a component for a different version, or substituting a component with similar functionality from a different vendor.
- Decoupling between components. There must be minimal coupling between components. Coupling can involve both functional coupling, such as procedure calls, as well as other dependencies such as resource contention or architectural assumptions. The architecture must allow for the isolation of components.
- Hiding unwanted functionality. In order to differentiate their product from competitors, COTS vendors often overload their systems with a large amount of functionality. Far from being an advantage in the COTS based system, the system architect may wish to remove this functionality. Since this cannot be done with a COTS component, the architecture must provide designers with a mechanism for masking the unwanted functionality so that it is inaccessible to the end-users and/or the system programmers.
- Debugging and testing. Since COTS components are black-box it is impossible to access their internals for the purposes of testing or debugging. An architecture and design cannot eliminate this problem, but it can include the capability of monitoring and verifying component behaviour during runtime, and preventing faults in a component from propagating through the system.

3. Example of COTS Integration

In order to better understand the issues relating to building system from COTS components we have conducted a number of experiments in building such systems. The most significant one that we have undertaken, and the one used in this paper to illustrate the architectural issues, is a distributed imagery management system. The capabilities of the system include the following:

- The system is capable of storing and retrieving various types of media (photographs, video, sound, etc.) Some of these artifacts will be stored in digital format while others will be stored as physical artifacts in a library.
- Artifacts will be stored at sites that are physically distributed. Some of the artifacts are replicated at different sites.
- A catalogue will be available on-line of all the artifacts. Each distributed site will have its own catalogue of locally held artifacts.
- Users will be able to electronically order physical copies of the artifacts, to download artifacts that are digitally available, and to replicate catalogue records when artifacts are downloaded or ordered.
- Special hardware/software (scanners, digital cameras, etc.) will be provided at workstations to generate digital artifacts and to store and catalogue them for later retrieval. Other workstations will be used for product preparation. These workstations will access on-line artifacts and process them to create products such as pamphlets, brochures, multimedia displays, etc.

The physical layout is shown in Figure 1. The system is constructed using a client server model. Server sites contain the cataloguing information and the artifacts in digital form. Clients communicate with the server in order to store, catalogue, search and retrieve artifacts. Servers communicate with each other to replicate artifacts and database records. All communication in the system is through standard internet protocols. Clients access the server through web interfaces.

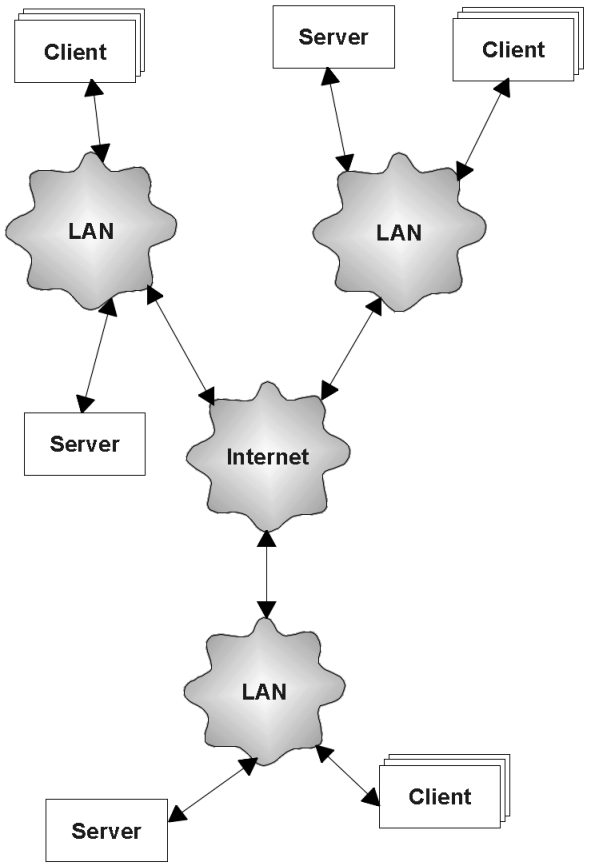

Figure 1.
Client-server model.

3.1 Catalogue Server Architecture

The catalogue server stores all the cataloguing information for locally held artifacts. The architecture of the servers are shown in Figure 2. Requests are received from the clients by the web server and passed on to the gluecomponents. The glue invokes local components to perform functions such as image conversion and database storage.

The various servers of the system have similar functionality but very different performance and resource requirements. One server is the main repository and contains replicas of most of the artifacts. This server is available to the entire user community. Other servers will be running on local desktop machines and be used by only two or three people within the department. The server architecture, and as many components as possible, should be portable across this wide range of platforms.

The COTS components that have been used to build the servers include the following:

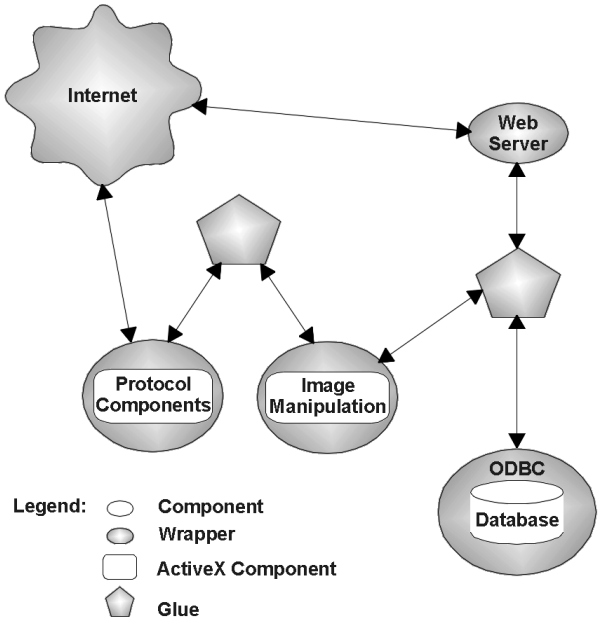

Figure 2.
Server side architecture.

- Databases. The database must be ODBC (or eventually JDBC) compliant. This allows us to use desktop databases such as Microsoft Access in the local server setups while substituting enterprise databases such as Oracle or Sybase in the main repository. We then can use identical code on both servers for database access.
- ActiveX components. We use ActiveX components to perform some well defined operations such as protocol implementation and simple image manipulation. Using ActiveX restricts us to the Win32 platform. However, by appropriately wrapping these components we hope to be able to configure the system for different platforms by substituting the appropriate platform specific components. This will require little or no changes to the other software components of the system.
- Object libraries. We use object libraries to build much of the middleware, particularly the CGI scripts. Most of the middleware is written in Perl so the object libraries are not strictly COTS as we do have access to the source and they are distributed under the Gnu license. However we have treated these in a manner similar to COTS components.
- Web servers. We have restricted ourselves to standard HTTP and CGI in order to be compatible with any of the web servers currently on the market.

Two principles we have tried to follow in the design of the system, and which will be discussed further in Section 4, are the use of wrappers and glue. Wrappers are code that we design and implement and provide the only allowed access method to the wrapped component. Glue is the middleware that manages the integration of the components. The wrappers surrounding components and the glue integrating components are shown in the server architecture of Figure 2.

Access to the COTS components is accomplished through open and standard interfaces where such standards exist. This

includes standard Web protocols, such as HTTP and CGI, and standard APIs such as ODBC. Where such a standard does not exist, as in the case of many of the ActiveX components, a wrapper is built around the component with an interface that we can control.

The wrappers and glue on the servers have been written primarily in Perl, with some use of Visual Basic. By using Perl (and using it carefully) much of the middleware glue can be ported to new platforms simply by copying the scripts and configuring the server appropriately.

3.2 Client Architecture

A basic client consists of a standard web browser. Through the use of HTML forms this allows for searching and retrieving information from the database, submitting work orders, etc. For this type of client the only commercial component required is the web browser and the underlying infrastructure (operating system).

Further COTS component integration will be required for clients that have specialized hardware and/or software. This includes clients that create and catalogue artifacts or production workstations where a set of artifacts is processed into a final product. For example, a production workstation will require a software application to process images and generate reports and brochures. Numerous applications exist for this purpose and one (or more) can be used on the workstation. We do not restrict which application a client should use.

Users should be able to interact with the servers and with the local software packages in an integrated and seamless manner. For example, once a set of items has been located by a catalogue search the user should be able to download these artifacts directly into a project in the product preparation application.

An example architecture for a client is shown in Figure 3. There are two major commercial components to the system: a web browser; and an application for developing the product. The user, through the browser interface, searches for the desired artifacts which may be stored locally or remotely. Once they are found, the artifacts are downloaded into the product preparation application. The user then interacts with this application's interface to generate the required product.

Integration code is responsible for receiving the artifacts from the server, opening a project in the preparation application, and adding the downloaded artifacts to this project. A wrapper is placed around the preparation application so that it has a standard and known interface. By writing wrappers for applications from different vendors the system can be configured with different processing applications based on the preferences of the user. Subsequent substitution of any component involves modification to the wrappers only and not to the COTS component nor to the glue.

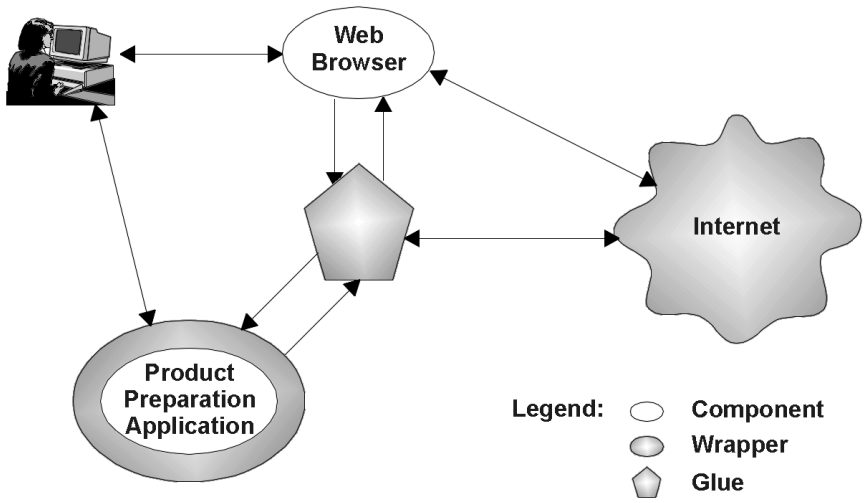

Figure 3.
Client side architecture.

4. Strategies for Integration

The purposes of a software architecture are to define the major components of a system, how the components interface with each other, and the interactions between components to provide the system services. In a COTS based development approach many of the components will be commercial components. This puts some immediate constraints on the architecture:

- The architecture must adapt to the connectors available in the components being used. For older and legacy components this could involve such primitive techniques as screen-scraping and terminal emulation; more modern components may export interfaces available through ActiveX, JavaBeans, or CORBA.
- The architecture must adapt to the functionality available in the components. This includes adding functionality that is desired but not included in the component, as well as hiding functionality that is included but the designer wishes to mask out.
- The architecture must adapt to the possible different data models and data formats used by the diverse components. This involves a process of data mapping or translation to ensure appropriate communication channels between COTS components

Components as received from the component supplier do not, in general, conform to the architectural requirements of the integrator. The functionality available in the component does not correspond to the functionality the integrator wishes to see in the component; and the interface to the component is non-standard or does not conform to what the integrator wishes. In order to minimize the architectural mismatch found in the component the integrator must adapt the component to the desired architecture while simultaneously adapting the architecture to the COTS components available.

While the majority of the systems components will be COTS-based, there will inevitably be a need to design custom components that interact with the COTS. These components provide added functionality for which a suitable

commercial application cannot be found. The design of these components can follow traditional design concepts but, in order to ensure a consistent architectural approach, these custom applications should be built so that they can be integrated into the system in a manner consistent with the integration of the COTS components. This includes the use of wrappers and glue, and consistency with the architectural style of the COTS components.

In order to design a system primarily for integration, we have identified a number of software integration components. These integration components serve different purposes and each has a different relationship to the components being integrated. The integration components that we use are: wrappers; glue; and tailoring. These are described in the following sections.

4.1 Wrappers

A wrapper is piece of code that the integrator builds to isolate the underlying COTS component from other components of the system. There are a number of reasons why a system architecture should include wrappers around components:

- Conform to standards, e.g., CORBA wrappers around legacy systems [1,7,9,16].
- Reduce the impact to the system of changes to the wrapped component.
- Provide a standard interface to a range of components. For mature domains a wrapper may be provided by the component supplier, e.g., ODBC. Standard interfaces are a prerequisite for substituting COTS components from different vendors.
- Add (or hide) functionality of a component.
- Give system integrator control over the "look and feel" of the component. Even though the integrator has no control over the component, the integrator does have access to the source and control over the wrapper.
- Provide a single point of access to the component.

One example of wrappers in the distributed imaging system is the ODBC interface to the Microsoft Access database. ODBC provides an industry standard API for accessing relational databases. By restricting access to the database to be ODBC compliant we can configure a system with a different database product by maintaining a similar data schema.

Another example of a wrapper is the script around the product development application. This wrapper provides a standard for moving data into the application. Unlike ODBC, which is an industry standard, this interface will be controlled by the designer. It will allow systems to be configured with different product development applications and for new applications to be plugged in as they become available.

4.2 Glue

The glue code provides the functionality to combine the different components. Purposes of the glue include:

- Control flow. Invokes functionality of the underlying components as required.
- Component bridge. Glue code can resolve any interface incompatibilities between components, for example by performing any necessary data conversions.
- Exception handling. By trapping exceptions the glue code can provide a consistent exception handling mechanism.

Within the distributed imagery management system, the server side glue code has been developed in Perl; on the client it is being developed in Visual Basic. In both cases they serve primarily as middleware combining components to add system functionality. For example, on the server there is a script that receives an image and cataloguing information from a client, invokes a component that converts the image format and creates an image thumbnail, stores the thumbnail and image,

and updates the database. In the current configuration the database is Microsoft Access made available through ODBC; the image manipulation software is an ActiveX component with a Visual Basic wrapper to provide a standard interface; and operating system calls are used for file access and manipulation. The glue code has been developed in Perl and, with the exception of the ODBC which is specific to Win32, is platform independent. Use of JDBC is planned in order to make the code truly platform independent.

An example of a glue component on the client side is the middleware between the local browser, remote database, and local product preparation application. This component performs the following actions: inputs the found set and downloads the images in the set; opens a new project in the product preparation application; moves the images into this project; opens the GUI to the product preparation application on the users desktop. This glue is being written as a Visual Basic component.

4.3 Component tailoring

Component tailoring refers to the ability for system integrators to enhance the functionality of a component in ways that are supported by the component vendor. The tailoring is done by adding some element to the component to provide it with functionality not provided by the vendor. Tailoring does not involve modifying source code of the component.

Examples of tailoring include "scripting", where an application can be enhanced by executing a script upon the occurrence of some event. Early versions of scripting include simple macro languages. The scripting capability in many newer applications has become more sophisticated with full-fledged programming languages and interpreters, such as VBA, tcl and Perl, being accessible within applications.

Another example of a tailoring capability is the use of plug-ins. A plug-in is a component that registers with the enclosing application. The enclosing application makes a call-back to the plug-in when its functionality is required. By publishing the registration and call-back techniques, COTS vendors provide COTS users with a method of enhancing the component functionality without access to the source code.

When tailoring components in this way, designers must remember that the tailoring aspects are components in their own right. The designer must treat them as separate configuration items, make sure they are installed with the corresponding container component, and make sure that they are carried along during the upgrades.

Although we do not currently use tailoring in the distributed imaging system, one potential use is with plug-ins for the web browser. Browsers can display only a limited number of image types (typically GIF and JPEG) and do not provide editing capability for these images. By using plug-ins for Netscape or Microsoft browsers we can enhance their functionality to display more types of image formats and allow users to markup and annotate the images from within the browsers.

5. Rules for integration

Having developed a number of systems that use off-the-shelf components we have begun to develop a set of rules-of-thumb for easing the task of COTS component integration. Through experience we have found that complying with these rules simplifies the task of development and evolution of systems. These rules are outlined and discussed in the following sections.

5.1 Wrap all components

Rule number one states that all off-the-shelf components should have wrappers placed around them. Many of the following rules depend on such wrappers being in place.

The justification for wrapping all components is that the wrapper provides the only mechanism by which the integrator can control the interface and interactions of the component, and isolate other components of the system from

changes to the underlying off-the-shelf component.

In addition, any custom software which is used to provide significant functionality in the system should be treated in a manner similar to a COTS component and integrated into the system using a wrapper. This will allow us to more easily substitute a COTS component for the custom component should future commercial developments provide the required functionality.

There are numerous examples in our work of where we have wrapped components. One example, on the server side of the image processing system, is the image manipulation component. This component is used for simple image manipulation functions, primarily format and size conversions. When looking for COTS components to provide this functionality we noted the following points:

- There are a wide range of possible solutions, each with its own interface. Even those within a single technology standard (e.g., ActiveX) had widely divergent interfaces.
- The functionality available in most potential COTS components far exceeded the functionality we required from the component. Masking out the unneeded functionality is therefore a significant concern.

By wrapping the component in a Perl module and exporting the appropriate objects we exported only the functionality of the component that we wished other programmers to use. It also allowed us to control and standardize the interface so that a substitution could be performed on the underlying component with no impact on other components of the system.

A different example of wrapping can be found in the client side of the imaging system. Certain clients contain specialized applications for product preparation and manipulating images. Users at these client stations should be able to find and download images from the database(s) directly into a project accessible by this application. We do not wish to state a priori what this image manipulation application is, and indeed want to support many such programs and leave it to the user to determine his/her favorite. Since there is no standard interface to these applications we needed to define and control the interface. The interface we have defined is primarily used to open a project and move the images from the network into the project. Once the project is established the image manipulation application is invoked and the user deals with the (familiar) interface of the application. In this case the wrapper is responsible only for the invocation of the COTS application and the establishment of the project files.

5.2 Make glue independent of underlying components

The glue provides functionality such as data and control flow, exception handling, and data conversion. Glue should be independent of the underlying components and should not change as different underlying components are substituted. Component changes should be hidden by the component wrapper and tailoring. Glue code only interacts with COTS components indirectly through the wrapper.

The functionality provided by the glue code should not depend on the specific off-the-shelf component that is being accessed. Services such as exception handling and control flow should evolve independently of the underlying components being glued together. As the components evolve and are modified, the glue need not be changed. By providing insulating wrappers around the components the glue can use standard methods for accessing the components.

In the image system, there are a number of examples of glue that is independent of the components. In the server system, the glue middleware is responsible for receiving images over the internet, using the image processing component, and inserting the image and cataloguing information into the database. By having standard interfaces to the imaging component and the database different components with similar functionality can

be substituted by conforming to the standard interface without modifying the glue.

A similar example can be found on the client side for the glue that downloads images from the internet and places them in a project in the production application. A wrapper providing a standard interface to the production application allows for this glue to remain the same when a new production application is substituted.

5.3 Verify component version compatibility.

Components, particularly COTS components, evolve rapidly with frequent releases of new versions. Systems often have dependencies as to which versions of components will operate together. If a particular component is upgraded to a new version, this frequently means that wrappers and other components must be upgraded as well.

Designers and implementors should verify, whenever possible, that the current configuration of components is version compatible. Ideally this version checking is automated. The verification can be done either at build time, if all components are bound together at build, at installation time, or at run time if late binding is used between components.

Both Perl and Visual Basic using ActiveX components have some support for version verification. With Perl, this is done by the component developer including a version number with each released module. The component user specifies a required version number when linking to the module. A run time check is performed to verify that the module being linked has a version number equal to or higher than the required version.

ActiveX components have both a component version and an interface version for each component. The component version is allocated by the component developer. It is used by the installation utility to determine whether a currently installed version of a component should be replaced with a newer version.

The interface version of a component is generated automatically when the component developer compiles the component. It records the compatibility between the interfaces of different versions of a component. If a component developer is creating a new version of a component with an interface incompatible with the previous version, a warning is given. At run time, when a Visual Basic program links to the ActiveX component, a check is automatically performed to verify that the interface exported by the component is the one expected by the Visual Basic program. The integration programmer can trap this exception if incompatible versions are installed.

5.4 Add assertions to the wrappers/glue

We are finding it useful in many cases to provide a high level of checking within the wrappers and glue in order to verify run-time assertions. The assertions can be as simple as verifying parameter types or values, or more complex, asserting relations between data values or required temporal orderings of events. Since the components are black-box, the glue and wrappers are the primary means by which developers can perform this type of run-time checking. By placing assertions between the components and raising an exception if they are violated, faults can be quickly detected and isolated within the system.

A simple example in the client software is the use of an assertion in the product preparation application wrapper that verifies the data type of the image being passed can be imported by the application. If a server provides image data in a format not recognized by the application (and we do not know a priori all the image formats the servers will provide nor all the product preparation applications that clients will use) an exception can be raised immediately and the user notified of the problem.

5.5 Do not have components talk directly to each other

Off-the-shelf components should always have some wrappers and glue between

them and not interact directly with each other. This allows some tolerances in how precisely the components must fit together and minimizes the coupling between components. As a components evolves, the wrappers and glue around the component can be updated but there is minimal impact on other components of the system.

A second reason for placing integration components between all COTS components is that this integration code is the only source code to which the developer has access. If there is any requirement for developers to add to the system extra capabilities for testing, debugging, fault isolation, or instrumentation, it must be done inside the glue and the wrappers[13]. Even in the case where a particular COTS component could interact directly with others in the current system, future versions may exhibit different characteristics.

5.6 Be compatible with open standards

Selecting COTS components that depend on proprietary standards can cause problems with portability of the systems and with configuring a system with components from different vendors. With closed standards developers become locked into a specific vendors product and cannot move to another vendors product even if there are other products with similar functionality. As the different software domains mature, more and more standards are being evolved in the different application domains, and market pressure often forces vendors to comply with these standards. Pulling in the opposite direction, vendors develop proprietary and closed standards with added functionality to give them a competitive advantage and lock in customers to their product.

Within the imaging system we have tried to conform to open standards wherever possible and avoid closed proprietary standards. This assists in providing plug-and-play of components from different sources, and in configuring the system for different requirements. For example, we adhere to the standards specified in HTTP, HTML and CGI. This allows us to easily configure the system for any Web browser and server.

5.7 Avoid early commitment to an architecture

If system integrators wish to take advantage of available COTS components, many of the architectural and design decisions must be taken concurrently with the component selection process. By committing to a specific architecture and specific technologies too early in the process the system developers may make the integration of components difficult or preclude the use of COTS components entirely.

6. Conclusions

COTS software components have been used by software developers for many years, but with the increased complexity of software systems and the new and emerging technologies this trend is increasing dramatically. Although COTS software integration is being done, the approach has tended to be ad hoc resulting in systems that are error prone and difficult to maintain. By carefully defining the architecture and design of the system many of the problems and issues raised by the use of COTS software components can be addressed within a more formal and defined process resulting in more reliable software that can evolve over time. This paper has presented the elements of an architecture for integration and defined some informal rules that facilitate this integration.

Acknowledgments

This research was supported by the Chief, Research and Development (CRAD) of the Department of National Defence.

About the Authors

Dr. Mark Vigder is a Research Officer with the National Research Council. He has a Ph.D. in Computer and System Engineering from Carleton University, Ottawa, and has twenty years experience

with software engineering as a practitioner, researcher, and educator.

References

[1] R.C. Aronica and D.E. Rimel Jr. Wrapper Your Legacy Systems. *Datamation*, 42(12):83-88, June 1996.

[2] T. Bao and E. Horowitz. Integrating Through User Interface: A Flexible Integration Framework for Third-Party Software. In *Proceedings: The Twentieth Annual International Computer Software And Applications Conference,* pages 336-342, IEEE Computer Society, Aug. 1996.

[3] A.W. Brown and K.C. Wallnau. Engineering Of Component-Based Systems. In *Proceedings of the 1996 2nd IEEE International Conference on Engineering of Complex Computer Systems*, pages 414-422, 1996.

[4] O.A. Bukhres and J. Chen and W. Du and A.K. Elmagarmid. *InterBase: An Execution Environment for Hetrogeneous Software Systems*. IEEE Computer. 26(8)57-69, Aug. 1993.

[5] J.C. Dean and M.R. Vigder. System Implementation Using Off-the-shelf Software. In *Proceedings of the 9th Annual Software Technology Conference*. Department of Defense, 1997.

[6] D. Garlan and A. Robert and J. Ockerbloom. Architectural Mismatch or Why it's hard to build systems out of existing parts. In *17th International Conference on Software Engineering*, pages 179-185, 1995.

[7] D.L. Moniz. Integrating Legacy Databases Into a Common Infrastructure Using CORBA. In *Proceedings of the 9th Annual Software Technology Conference*. Department of Defense, 1997.

[8] J.K. Ousterhout. *Scripting: Higher Level Programming for the 21st Century*. Unpublished manuscript, available at http://www.sunlabs.com/people/john.ousterhout/scripting.html.

[9] C.M. Pancerella and R.A. Whiteside. Using CORBA to integrate manufacturing cells to a virtual enterprise. In *Proceedings of the SPIE Volume 2913*, pages 148-173, The International Society for Optical Engineering, 1997.

[10] M.Shaw and D.Garlan. *Software Architecture: Perspectives on an Emerging Discipline,* Prentice Hall Publishing, 1996.

[11] K.J. Sullivan and J.C. Knight. Experience Assessing an Architectural Approach to Large-Scale Systematic Reuse. In *18th International Conference on Software Engineering*, pages 220-229, Berlin, 1996.

[12] M.R. Vigder and W.M. Gentleman and J.C. Dean. *COTS Software Integration: State of the Art*. National Research Council of Canada, Institute for Information Technology technical report NRC39198, January, 1996.

[13] J. Voas and G. McGraw and A. Gosh. Gluing Together Software Components: How Good is Your Glue. *In 14th Annual Pacific Northwest Software Quality Conference*, pages 338-349, Oct, 1996.

[14] L. Wall and T. Christianson and R.L. Schwartz. *Programming Perl 2nd ed*. O'Reilly & Associates, Inc. 1996.

[15] *Microsoft Visual Basic Programmer's Guide*. Microsoft Corporation, 1995.

[16] Common Object Request Broker: Architecture and Specification. Object Management Group.

[17] *USENIX Very High Level Languages Symposium Proceedings*. USENIX Association, Berkely, California, 1994.