Architectural Evolution of Legacy Systems

George T. Heineman
Computer Science Department
WPI
Worcester, MA 01609
heineman@cs.wpi.edu

Alok Mehta
Vice President
American Financial Systems
Weston, MA
amehta@afs-link.com

WPI-CS-TR-99-05
February 1999

Keywords – Software Architecture, Components, Active X, Life Insurance and Executive Benefit Software, Microsoft Visual Basic, Component Specification Language (CSL).

Abstract
The purpose of this paper is to gain experience in solving real problems faced by a company. We first specify the system architecture of the AFS Master System ® using our Component Specification Language (CSL). We then identified various problems evident in the current architecture of the AFS Master System ®. Based on an analysis of the architecture and these problems, we proposed a modification to the software architecture that addressed five out of the seven main problems identified. The engineers made the appropriate changes to the software system (about one week of effort) and have noted a 25% improvement in efficiency as well as an improved system organization that can be more easily changed to meet future demands. We believe the type of architectural change described in this paper will prove useful to developers using similar technologies as described in this paper.

1. Introduction

The emerging discipline of Software Architecture, as defined by Garlan and Shaw, is concerned with a level of design that addresses structural issues of a software system, such as: global control structure, synchronization, and protocols of communication between components [1]. Software Architecture is thus able to address many issues in the development of large-scale distributed applications using off-the-shelf components. In particular, it is a useful vehicle for managing *coarse-grained software evolution*, as observed by Medvidovic and Taylor [2]. However, recent approaches to architectural evolution, such as ArchStudio [10], focus on evolving systems that are already designed and constructed from well-defined components and connectors. This paper applies Software Architecture results to a legacy system.

We selected the AFS Master System® (AMS) for our case study since we knew that American Financial Systems (AFS) was unsatisfied with certain aspects of their existing application. The primary business objectives for AFS regarding AMS are improving the ease of use, performance, and reliability of AMS. We first specified the system architecture of AMS using our Component Specification Language (CSL) [3]. This exercise proved useful since it revealed certain extensions necessary to CSL (which, for lack of space, we will not present in this paper). We then identified various problems evident in the current architecture of AMS. Based on an analysis of the architecture and these problems, we proposed a modification to the software architecture that addressed five out of the seven main problems identified. The engineers made the appropriate changes to the software system (about one week of effort) and have noted a 25% improvement in efficiency as well as an improved system organization that can be more easily changed to meet future demands.

We believe this paper is relevant since it describes the evolution of a software system that incorporates technologies such as Microsoft Visual Basic, Windows NT, and ActiveX components. Section 2 contains the overall methodology we suggest for architectural evolution of legacy systems. In Section 3, we describe the current architecture of AMS. Section 4 describes the main architectural problems identified by AFS, and Section 5 presents the modified system architecture. We close the paper with discussions of related work and our conclusions.

2. Methodology

One of the most difficult issues with legacy systems is that as they evolve over time, the complexity of the system increases [11]. Changes to a localized component must be shown not to disrupt the global communication between system components. As more components and features are added to a system, it is imperative that the communication protocol between system components be maintained and accurately documented. However, often the only architectural documentation available is a static representation of system components and their relationships. The Software Architecture community has developed a framework composed of components and connectors for describing software systems [1]. While components can be identified in straightforward fashion, often connectors are elusive since the code to communicate between components is often embedded within the components themselves. We suggest the following four-step approach that we have pursued in the case study described in this report.

2.1 Identify components

The components of a software system can be stand-alone executables or software modules. The primary focus of this task is to identify the public interfaces that define the allowable communications between components. This may include any of the following: public method interfaces, global variables, shared memory, shared file system, network connectivity, and database systems.

2.2 Identify communication between components

Once the individual components have been identified, the next step is to capture the communication channels between the components. Most architectural diagrams with boxes and lines are sufficient for capturing the binary relationship that component C_1 communicates (in some fashion) with component C_2. There often are multiple channels between C_1 and C_2, however, and each must be clearly identified and described.

2.3 Identify Connecting-Components

When components directly communicate with each other, there is increased *coupling* between them. This is undesirable from a design perspective since a change in one component may force a compensatory change in the other component. Many approaches have been developed over the years to address this problem, such as component adaptors [13] and object-oriented design patterns[14]. The legacy system may also have its own individual solution. When two components communicate through a third component C_3, C_3 is called a *Connecting-Component*. The benefit of having C_3 is that the communication between C_1 and C_2 can evolve more flexibly than if C_1 and C_2 were highly coupled. These connecting-components are analogous to *connectors* [1], but the main difference is that connecting-components have *ports* since they are components, while connectors have associated *roles*.

2.4 Evolve communication

To change the communication between components C_1 and C_2, we first introduce a connecting-component, C_3, if one doesn't already exist. Then, the evolution of communication can occur independently between (C_1 and C_3) and (C_2 and C_3). The individual legacy components will require certain modifications to enable this change, but in the future, architectural evolution will be easier to accomplish.

This methodology is a good strategy that should be followed whenever a legacy system undergoes change because it: 1) is incremental; 2) improves the architectural integrity of the legacy system by replacing implicit communication between system components with explicit, documented connecting-components; 3) results in a better documented architecture. We now apply this methodology to AMS.

3. Current Architecture of AMS

Figure 1.0 provides an overview of the AMS architecture, provided by AFS engineers, and the relationships between its constituent components. The following section provides a detailed analysis of each component, communication vehicle, type of communication, and public interfaces.

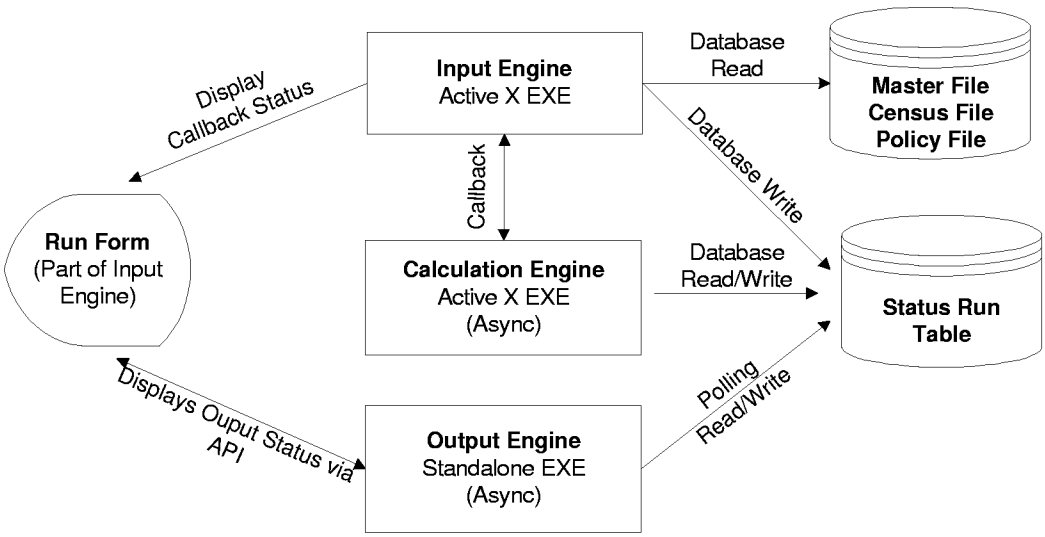


Figure 1: Overview of current Architecture

3.1 The Input Engine, Calculation Engine and Output Engine

There are three primary components that constitute AMS: the Input Engine, the Calculation Engine and the Output Engine. The Input Engine and Calculation Engine are Active X executables; the Output Engine is a stand-alone executable. Microsoft (MS) Access ® is used as data repository both to manage user data and to act as a communication vehicle between the three engines. Active X is part of Microsoft's COM (Component Object Model) [4] technology. There is a Print Engine, which we omit for space reasons, that simply delivers the reports generated from the Output Engine to a printer.

The Input Engine manages input data and prepares user data for the Calculation Engine. User data is stored in Master and Census files (MS Access ® tables). The Master File contains plan level information about life insurance while the Census file contains policy and individual level information. A plan can have many individuals and an individual can have many policies. A case is defined as a combination of Master and Census Files. The user initiates a Case (or a "Run") after entering a series of parameters. The Calculation Engine is then invoked by the Input Engine and it stores its calculations in an MS Access ® Table. Through the MS Windows ® API (Application Programming Interface) [1] and polling mechanism (see Section 3.3.2), the Output Engine generates and displays reports to the user. Figure 2 illustrates the user interface. The run status displayed in the Run Form shown in Figure 3, is continually updated to reflect the status of the ongoing "Run".

3.2 Communication Vehicles

There are two main communication vehicles between these engines: Status Run Table and Run Form.

3.2.1 Status Run Table

The Status Run Table, created by the Input Engine, contains information about the progress of the calculation and the printing of reports. When a calculation is complete, the Calculation Engine updates the status of the record representing that calculation to a "6". When the Output Engine reads (via Polling) a status of "6", it generates a report and updates the status to "14" when done. The communication between the Output Engine and the Status Run Table occurs via polling (see Section 3.3.2). There are many records in the Status Run Table in a given session since both the Calculation and Output Engine operate asynchronously.

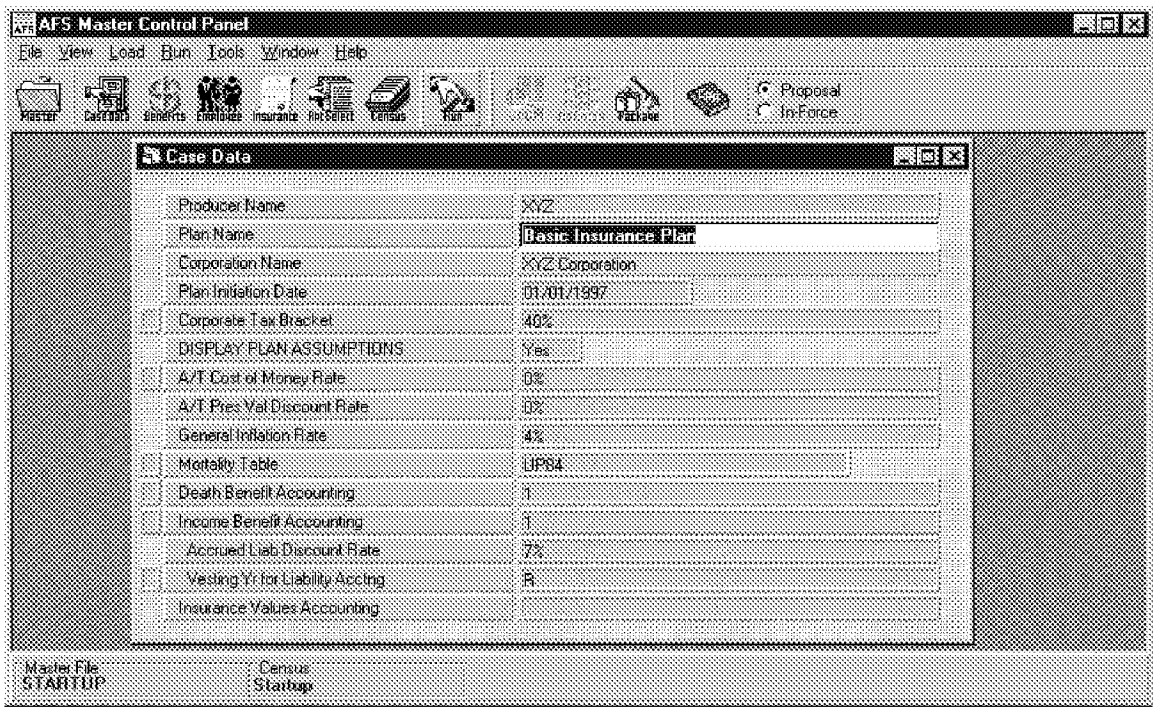


Figure 2: Screen Shot of Input Engine

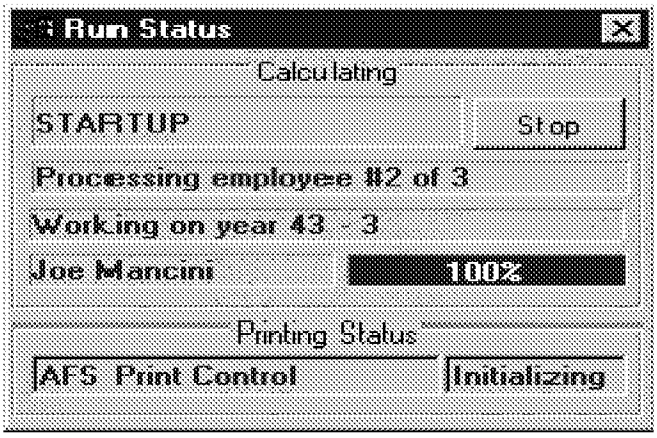


Figure 3: Run Form

3.2.2 Run Form

A Visual Basic application is composed of a set of *Forms* that are the windows with which a user interacts when running the application. Forms have properties, event listeners, and methods that control their appearance and behavior. The Run Form is part of the Input Engine. The Calculation Engine communicates with the Input Engine through a callback mechanism (or callback for short) [6]. The Run Form displays messages sent from the Calculation and Output Engines using callbacks. However, since the Output Engine is a stand-alone executable, it can only send messages to the Run Form via the Windows API. Figure 3 illustrates the Run Form. The text boxes in the Run Form are used to display the case name, employee name, calculation status, and printing status. The Stop Button will prematurely interrupt and terminate the Run when pressed.

3.3 Types of Communication

Essentially, there are three types of communication between the three engines: Callback, Polling, and Application Programming Interface (API).

3.3.1 Callback

A Callback construct decouples components so they can communicate without knowing in advance with whom they will be communicating. The Calculation Engine supports a COM interface of *Connect*, *Disconnect*, *Interrupt*, and *Run*. The *Connect* method passes a Callback object to the Calculation Engine and the *Disconnect* releases it; these two methods are similar to the *addEventListener* and *removeEventListener* methods featured in JavaBeans [7].

3.3.2 Polling

Polling decouples the control flow between two processes that require only intermittent communication. The Output Engine is initiated by a simple API call (FINDWINDOW), performed by the Input Engine. Once initiated, the Output Engine continually polls the Status Run Table for instructions. This data table is created and updated by the Input Engine and by the Calculation Engine's Callback methods. Once the Output Engine receives instructions to print an individual, it asynchronously processes the queues of Tables stored by MS Access ® (suffixed by the letters A, B, C, E, Y or F) created by the Calculation Engine.

3.3.3 API

In the current architecture, API calls are used to send messages and trigger events between the Input Engine (Run Form) the Output Engine. These API methods are probably not intended to be use for inter-process communication, so programmers must be careful to document their every use in a software system. The Input Engine is responsible for telling the Output Engine to process the data in the MS Access® tables. The Input Engine uses a Windows API method POSTMESSAGE to generate a double-click event within the Communication Form. Another example of an API occurs when the Calculation Engine passes text to the Run Form through a similar process. In this case, the Calculation Engine first changes the caption of the hidden Communication Form (part of the Output Engine) to the specific text it desires to send; it then uses the POSTMESSAGE method to send a double-click to the Run Form which in turn reads the caption and displays it to the user.

3.4 Public Interfaces

We now define the public interfaces for the three primary system components: the Input, Calculation, and Output Engines. The Input Engine is the main component, as well as the central messaging entity. Figure 4 contains the important public interfaces of the Input Engine. The public interface for the Calculation Engine is outlined in Figure 5. Figure 7 describes the important public interfaces of the Run Form. Since the Output Engine is not an Active X Executable, it does not have any public interfaces. However, it communicates with the rest of the System via API calls and polling. In Figure 6, we define the API calls that are used by the Output Engine.

Interface	Parameters	Purpose
Callback	SERVER_CODE as Integer RUN_ID as Long EE_ID as Long EE_NAME as String POLICY_ID as Long STATUS as Integer	Displays information about the on-going processing on the Run Form. The SERVER_CODE represents the caller (either the Calculation Engine or Print Engine). RUN_ID is a combination of user data that represents a case. EE_ID, EE_NAME and POLICY_ID represent the Employee's policy.

Figure 4: Input Engine Interface

Interface	Parameters	Purpose
Connect	OCB as Object	Connects the Calculation Engine to a Callback Object passed in as a parameter.
Disconnect		Releases the memory for the OCB object.
Run	RUN_ID as Long RUN_TYPE as Long	Loads the hidden Communication Form that asynchronously invokes the Calculation Engine's main processing routine. RUN_ID is a combination of Master and Census File from MS Access ®. RUN_TYPE represents the calculation mode.
Initialize	RUN_ID as Long	Sets global flags and initiates the connection to the database.
Interrupt	RUN_ID as Long	Stops the Calculation Engine from processing when called. It sets a global flag called *nInterrupt*, that causes the Calculation Engine to interrupt any further processing.

Figure 5: Calculation Engine Interface

Interface	Parameters	Purpose
DOUBLECLICK		When a double-click Event is received, this event listener starts processing the MS Access ® tables into reports.
GETWINDOWTEXT		Other components can retrieve Text that the Output Engine wishes to communicate by retrieving the Caption of the Communication Form Window.

Figure 6: Communication Form Interface

Interface	Parameters	Purpose
PRINTMESSASGECLICK	INDEX as Integer	When a double-click Event is received, this event listener retrieves the caption from the hidden Communication Form in the Output Engine and displays it in the appropriate text box as determined by the INDEX parameter.

Figure 7: Run Form Interface

3.5 CSL specification of current architecture

Figure 8 contains the CSL specification of the existing AMS architecture. One should first note the structural topology of the specification. AMS is composed of five components: *InputEngine, DatabaseEngine (Microsoft Jet Engine Version 3.51 library), CalculationEngine, OutputEngine,* and *Windows*. Each of the interfaces described in Figure 4 through 7 is represented in the CSL specification. This specification describes a slightly different architectural topology than the one shown in Figure 1. In particular, although the Run Form is marked as "Part of the Input Engine", Figure 1 attempts to show the Run Form as a separate entity. In addition, there is no discussion of the Communication Form, nor is there any discussion of how the various components use API calls to communicate with each other. A system's weakest points are often such undocumented parts.

4. Problems with the Current Architecture:

The previous section described the infrastructure of AMS. In this section, we identify several problems that exist with the current system architecture.

4.1 Starting, Re-starting, and Stopping the System is not always consistent

Many users have observed that re-running a case (see Section 3) produces inconsistent results. AFS has tried unsuccessfully to debug this problem. When the user wants to stop a run, the Stop Button on the Run Form is supposed to update the Status Run Table, which will interrupt the Run once the Output Engine reads the updated the Status Run Table. Because the Calculation and Output Engine operate asynchronously, AFS engineers have had a difficult time debugging the problem. They have attributed this problem to two things: non-determinism of Polling and an program in correctly updating the Status Run Table. In any event, this problem reveals the weakness of the current architecture.

```
interface callBackInterface {
 void Callback (int SERVER_CODE,
    Long RUN_ID, Long EE_ID,
    String EE_NAME, Long POLICY_ID,
    int STATUS);
}

object OCBCallback implements
 callBackinterface;

system AFSMaster {
 component Windows {
  port API extends InOutPort {
   void POSTMESSAGE (Event e);
   void GETWINDOWTEXT (Handle window,
                       Text string);
   void SENDMESSAGE (Handle window,
                     Event e);
   void SETWINDOWTEXT (Handle window,
                       Text string);
   Handle FINDWINDOW (Text string);
   Handle GETACTIVEWINDOW ();
  }
 }

 component InputEngine {
  component RunForm {
   port processAPI extends InPort {
    // handles API events
    void PRINTMESSAGECLICK (Event e);
   }
  }
  port callBack extends InPort {
   void Callback (int SERVER_CODE,
      Long RUN_ID, Long EE_ID,
      String EE_NAME, Long POLICY_ID,
      int STATUS);
   }
  }
 }
```

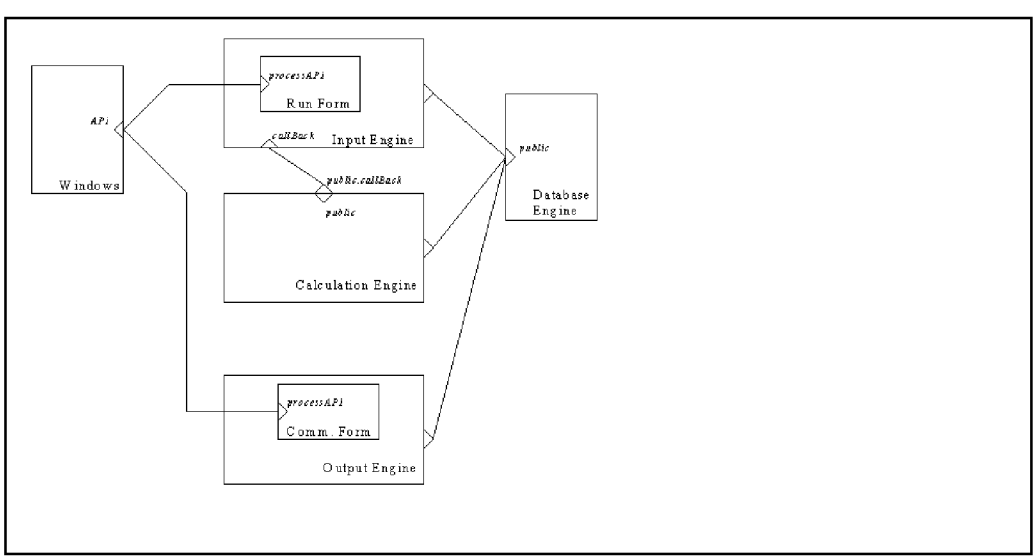
```
component CalculationEngine {
 port public extends InPort {
   void Connect (OCBCallback OCB);
   void Disconnect ();
   void Run (Long RUN_ID,
             Long RUN_TYPE);
   void Initialize (Long RUN_ID);
   void Interrupt (Long RUN_ID);

  port callback extends OutPort {
   void Callback (int SERVER_CODE,
      Long RUN_ID, Long EE_ID,
      String EE_NAME, Long POLICY_ID,
      int STATUS);
   }
  }
 } implemented by {
    boolean    nInterrupt;
 }

 component OutputEngine {
  component CommunicationForm {
   port processAPI extends InPort {
    void DOUBLECLICK (Event e);
    void GETWINDOWTEXT (Handle window,
                        Text string);
   }
  }
 }

 component DatabaseEngine {
  port public extends InPort {
   void DBREFRESHCACHE ();
  }
 }
}
```

Figure 8: CSL specification of architecture with graphical representation



4.2 Access Database Issues

Sometimes the MS Access ® database does not write to disk when it performs a COMMIT. As a result, the data generated by the Calculation Engine is not always refreshed into the relevant tables. When this occurs, the Output Engine does not receive reliable data. AFS engineers have decided that MS Access ® sometimes caches data rather than performing a write. In response, they have identified several places where they had to manually force the database to write its cache to disk using a special command, DatabaseEngine.DBREFRESHCACHE.

4.3 Inefficiencies from Polling

While the Calculation Engine is generating the necessary tables (with the suffix Y, A, B, C, E, V, R, and F) the Output Engine continually checks the Status Run Table for instructions to process those tables. Polling is inefficient, leading to wasted processing time and on occasion, the polling mysteriously fails.

4.4 Lack of Distributed Processing

The current architecture does not enable distributed processing for three reasons: the use of Windows NT API calls, the Output Engine is a stand-alone executable, and the Calculation Engine contains some user interface code. Since one of the strategic objectives for AFS is to web-enable AMS, the communication infrastructure must change to support some form of distributed processing.

4.5 System cannot be used in a multi-user environment

Currently, AMS cannot be run in a multi-user environment. Since the Input and the Calculation Engines are Active X components, it will not be too difficult to integrate them into a multi-user environment. The use of API calls will not work in a multi-user environment because there will no longer be a single instance of the Run Form. In addition, there must be some mechanism to uniquely identify the data belonging to each user.

4.6 Inefficiencies of the Output Engine

The Output Engine is often idle because of polling and the use of API calls when communicating with the Input Engine. Since the Input Engine and the Calculation Engine are Active X components, they are used only when needed. We believe that converting the Output Engine to an Active X DLL will not only improve the stability and control of the Output Engine but will also enhance its performance. This is because an Active X DLL is an in-process server [6] that resides within the same context as the Input Engine, unlike a stand-alone executable.

4.7 Double click from the Output Engine sometimes sends message to incorrect window

The Output Engine sends an API message (POSTMESSAGE DOUBLECLICK) that must reach the Run Form. Occasionally, another Form may be in focus, resulting in misdirected communication.

5. Revised Architecture: Creating the Output Engine as an Active X Component (DLL)

As we examine the problems mentioned in the previous section, we observe that if the Output Engine is converted to an Active X component, we can address problems mentioned in 4.1, 4.3, 4.5, 4.6 and 4.7. As we can see from our analysis in Section 4, there are problems using the API calls. First, we have less control over the calls. Second, API calls will not work in a distributed computing environment. Finally, the implementation is very clumsy and hard to maintain. By implementing the Callback mechanism and removing the API calls in the Output Engine, we will essentially convert the Output Engine from a stand-alone executable to an Active X component with a COM interface.

We could have chosen the Output Engine to be an Active X executable just like the Calculation Engine, but instead, we chose it to be an Active X DLL. Why? DLLs [6] are in-process OLE Servers that execute within the client memory space thereby running significantly faster than their executable counterparts. However, one can argue that the Calculation Engine should also become an Active X DLL. Currently, there are certain Custom Controls [6] that are part of the Calculation Engine, preventing it from being a DLL. This is a known issue and we are addressing it.

We now document the steps required to convert the Output Engine to an Active X component.

5.1 Remove all API calls (Declaration, Events and Actual Calls) from the Output Engine

We removed all API calls used by the Output Engine to communicate with the Input Engine. To be more precise, the following API calls were removed: GETACTIVEWINDOW, GETWINDOWTEXT, FINDWINDOW, SETWINDOWTEXT, POSTMESSAGE, and SENDMESSAGE.

5.2 Remove API Listener Methods

As described in Section 4, PRINTMESSAGECLICK in the Run Form is the main routine invoked when the Output Engine sends messages via API calls. We removed this routine from Run Form.

5.3 Add a mediator object shared by Input and Output Engines

In the Microsoft Visual Basic ® Programming environment, we simply used the Project Add Class Module menu item. In doing this, it was important to make sure that the instancing property of this class module is set to 5 – Multi use. This allowed us to create an instance of this class as an Object in another component (the Input Engine). This mediator object reifies the connector between the two components.

5.4 Implement Connect and Disconnect Interfaces for mediator object

These are very simple but powerful routines. The purpose of this step is to implement the basic COM interface in the "OutputEngine.CLS". The following was implemented:

```
Option Explicit                          'Forces to declare all variables
Public CurrentDirectory As String        'So that Output Engine can set the startup directory

Public Function Connect (oCB As Object) As Boolean     'oCB is the reference to callback
  ChDir CurrentDirectory
  Set oCallBack = oCB                    'Sets the Global Variable oCallback
  Connect = True
End Function

Public Sub Disconnect( )                 'Disconnects the Callback Object
  Set oCallBack = Nothing
End Sub
```

5.5 Declare a global Callback object in the Output Engine

By declaring the Callback object a public object in the Output Engine, any method in the Output Engine can call its (public) methods. The following lines were implemented in the global declaration section of the Output Engine:

```
Public oCallBack As Object
```

5.6 Create an instance of the Output Engine and pass the reference to the Callback Object

The following code was implemented in the Input Engine to ensure that it will correctly instantiate the Output Engine as an ActiveX component:

```
Dim Returns As Long
Set Output_Engine = New OutputEngine
Output_Engine.CurrentDirectory = CurDir$
Returns = Output_Engine.Connect(goCallBack)
```

Note that *goCallback* is the reference of the Callback Class (in the Input Engine) that is passed to the Output Engine. Also, note that *Output_Engine* is a public object in the Input Engine.

5.7 Program the Input Engine

The Input Engine must acquire additional functionality to interpret the status codes received through the callback interface; Figure 9 contains this list. The first four status codes are reserved for passing information such as EE_ID, CENSUS_ID, POLICY_ID and EE_NAME that remain constant throughout the life of the execution. These status codes will be displayed to the user on the Run Form. This programming was done in the Callback Class module of the Input Engine.

5.8 Program the Output Engine

Similarly, the Output Engine needs to be modified to convert every API call (that used to send mouse events) into a callback invocation. This programming was done in the *SetRunPanelManual* routine of the Output Engine.

5.9 Change the Project Type to an Active X DLL

The compiler property was set so that an Active X component could be dynamically created and loaded.

With the completion of these nine steps, we converted the Output Engine to an Active X component with a public interface of *Connect* and *Disconnect*. The modified architecture as specified by CSL is contained in Figure 10.

Status	Meaning
5	*Starting* - Tells Input Engine that the Output Engine is starting up
6	*Setting up* – Initialization of some routines
7	*Running* - Performing Reports, Header, Footnote integration
8	*Done* – Finished integrating reports etc.
9	*Cancelled* – Received a cancelled status from the Status Run Table
10	*Preview* – The Output Engine is previewing the report
11	*Case Printed* – requested case is printed
12	*Terminated* – Terminated by user request
13	*Initializing* – Initialization of the Output Engine
14	*Dormant* – Waiting for more instruction

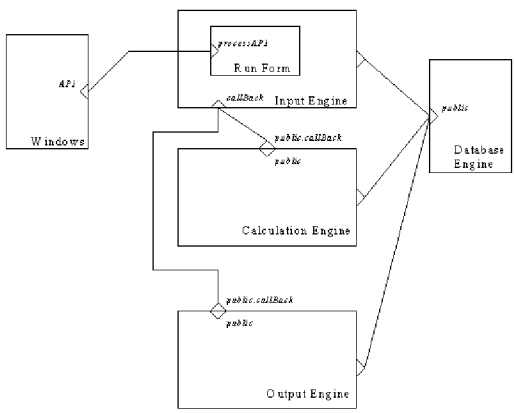
Figure 9: Status codes for callback functions

```
system AFSMaster {
 component InputEngine {
  component RunForm {
  }
  port callBack extends InPort {
   void Callback (int SERVER_CODE,
      Long RUN_ID, Long EE_ID,
      String EE_NAME, Long POLICY_ID,
      int STATUS);
   }
  }
 }
```

```
component OutputEngine {
 port public extends InPort {
  void Connect (OCBCallback OCB);
  void Disconnect ();

  port callback extends OutPort {
   void Callback (int SERVER_CODE,
     Long RUN_ID, Long EE_ID,
     String EE_NAME, Long POLICY_ID,
     int STATUS);
   }
  }
 }
}
```

Figure 10: Modifications to architectural Specification with graphical representation



6. Related Work

The work carried out as part of this case study is related to several efforts in the Software Architecture community. First, this paper contributes to the growing understanding that the process of specifying the software architecture for a software system has direct benefits. Many Architectural Definition Languages (ADLs) have been developed but there is still need to be more emphasis in solving industrial problems. Rapide, for instance, has been used to characterize the X/Open protocol for transaction management systems [8]; it is clear, however, that industry will only embrace the principles and technologies developed by the Software Architecture community when we reach out to solve problems of interest to industry.

A second contribution of this paper is the attempt to bridge the gap between research and industry. Many researchers propose techniques for managing change in the software architecture for a software system, but few have addressed legacy issues. *Architectural Reconfiguration* refers to the post-deployment replacement of a software component within a given software architecture. Andersson [9] and others suggest that reactive architectures can detect anomalous situations and, in response, automatically replace or reconnect its components accordingly. Such automatic approaches, however, will only work if a software engineer ensures that the software system operates correctly with the new components. *Architectural Evolution* refers to the architectural changes that occur within a system naturally as it evolves over time, but other approaches focus on replacing individual components or connectors [12]. Our paper seeks to evolve earlier, legacy systems so recent ideas on architectural evolution can be applied.

7. Conclusions

The intent of this paper was not to solve the problems of AMS, but to demonstrate that simple architectural analysis can lead to a more comprehensive understanding of a complex software system. In this experience report, we demonstrate a simple technique: abstracting the communication between two components into a connecting-component. Using this abstraction, we show an easy way to convert a stand-alone executable into an Active X Component (DLL). We also demonstrate that architectural analysis helps to achieve business objectives.

The main contribution of this report is to show that an architectural specification of an existing (legacy) system is important for verifying and improving global system properties, such as reliability. The conversion of the Output Engine was implemented in less than a week. The speed of execution of the Output Engine after being converted to an Active X DLL has improved 25% over its stand-alone executable counterpart. The communication between the three engines (the Input Engine, the Calculation Engine and the Output Engine) has improved tremendously. Thus, improving the reliability of the software. We are happy to conclude that this implementation successfully solved the problems mentioned in the sections 4.1, 4.3, 4.6 and 4.7. Being an Active X Component, the Output Engine now presents a better stage for making the system multi-user capable which in-turn lays an improved infrastructure for problems mentioned in the sections 4.4 and 4.5.

The AFS engineers are currently seeking to web-enable AMS, which will require more advanced architectural changes than those described in this paper. This effort will further test the ability of CSL to specify software architectures. In this way, we hope to continue our success at integrating software architecture research with the needs of industry.

References:

[1] David Garlan and Mary Shaw, An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering, volume I. World Scientific Publishing, 1993

[2] Nenad Medvidovic and Richard Taylor, Separating Fact from Fiction in Software Architecture, *Third International Workshop on Software Architecture*, Edited by Jeff N. Magee and Dewayne E. Perry, pages 105-108, Orlando, Florida, November 1998.

[3] George Heineman, Adaptable Software Components, *Twenty-second International Conference on Computers and Applications*, pages 121-127, Vienna, Austria, August 1998

[4] Richard Grimes. Professional DCOM Programming, Pages 25-27. Wrox Press Limited.

[5] Dan Appleman. Dan Appleman's Visual Basic 5.0 Programmer's Guide to Win32 API, Ziff Davis Press.

[6] Dan Appleman. Dan Appleman's Guide to Active X Components, Ziff Davis Press.
[7] Sun Microsystems, Inc. JavaBeans Specification 1.0, December 1996.
[8] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. *Specification and analysis of system architecture using Rapide*. IEEE Transactions on Software Engineering, 21(4):336-355, April 1995.
[9] Jesper Andersson, Reactive Dynamic Architectures. *Third International Workshop on Software Architecture*, Edited by Jeff N. Magee and Dewayne E. Perry, pages 1-3, Orlando, Florida, November 1998.
[10] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor, Architecture-based runtime software evolution. In *International Conference on Software Engineering*, Kyoto, Japan, April 1998.
[11] M. M. Lehman, Laws of Software Evolution Revisited, *Fifth European Workshop on Software Process Technology (EWSPT'96)*, pp. 108-124, Nancy, France, October 1996.
[12] Peyman Oreizy, Decentralized Software Evolution, *International Conference on the Principles of Software Evolution (IWPSE 1)*, Kyoto, Japan, April 1998.
[13] D. M. Yellin and R. E. Strom, Protocol specification and component adaptors, *ACM Transactions on Programming Languages and Systems*, 19(2):292-333, March 1997.
[14] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *Design Patterns: Elements of Reusable Software*, Addison Wesley, 1995.