Migrating Enterprise Applications to the .NET Framework

by S.K. Dutta

Abstract: This article describes the challenges faced during migration of an enterprise application product from ASP to the Microsoft .Net platform. Active Server Pages (ASP) has been a very popular means to develop web-based applications on the Microsoft platform for the past few years. The popularity of ASP stems from the fact that it is very simple to learn and use. For complex applications, however, the design patterns offered by the ASP-based programming made the code difficult to enhance, and as a result, maintenance became expensive. Microsoft .Net framework seemed to have filled in the gap by providing a framework that introduces object-oriented discipline in programming such applications. Also, many components were unavailable in the ASP framework (e.g. a good file upload component). These had to be purchased from third-party software vendors resulting in dependence on these components on the enterprise application. In addition, system services like authentication, encryption, etc. are in-built in the .Net framework. We will describe here a web-based application portal called *OfficeClip* and follow its progress as it is migrated in the .Net framework.

Product History

OfficeClip is a web-based enterprise application portal comprised namely of enterprise-wide time and expense reporting, project tracking, document management and calendar management, etc. It was initially designed using ASP at the front end and SQL Server at the back end. This choice was primarily made by considering the time to market and extensibility of the back end. Over the next few years, more applications were added to the *OfficeClip* suite, and the size of the sources became large (more than 300,000 lines of code). Enhancement and maintenance started becoming more and more expensive. Sometime during the middle of 2001, the decision was made to migrate *OfficeClip* to a platform that would provide easier maintenance and more room for growth.

Platform Selection

After evaluating various platforms, we narrowed down our choices to two enterprise platforms to migrate our application to: 1. Java, a matured platform developed by Sun Microsystems, and 2. .Net, an upcoming platform released by Microsoft (it was beta 2 at that time). The advantage of Java is that it is a proven platform for enterprise applications (J2EE standard) and has the ability to run seamlessly on both UNIX and Windows platforms. On the other hand, the .Net platform was not proven at that point, so we decided to look into it cautiously. The beta version of the .Net release was stable enough to convince us of the stability of our application if written in that platform. Also, looking into various components of the .Net platform, we felt that it addressed the problems we had with our ASP code. Because of its design philosophy, we understood that we would have more reuse if we used the .Net platform.

Migration Goals

Prior to beginning the migration process, we developed the following list of goals:

1. Provide three levels of customization to the end user, defined as follows:

    a. Level 1 - XML based configuration files for configuration of various application properties.
    b. Level 2 - Developer API to modify and extend the interface.
    c. Level 3 - Provide presentation layer source code to the client so they will be able to create or modify the existing presentation layer according to their needs.

2. Provide clear separation between various layers of the three-tier architecture, including the creation of a services layer to make it possible to provide and consume web services.

3. Provide protection of intellectual property rights.

4. Provide integrated security and authentication.

5. Provide the ability to use the product on mobile devices with minimal code changes.

6. Provide the ability to implement *OfficeClip* in various localized versions.

7. Provide the ability to create custom controls for reuse.

8. Provide the ability to compile code and distribute the executable, as opposed to distribution of the sources.

Most of the above objectives were met directly by the use of the Microsoft.Net platform. This framework has tools for creating web services, integrated security and authentication, support for localization, and has a mobile toolkit. The only thing we found it did not address very well was the protection of intellectual property.

The Microsoft.Net framework is designed to run on a runtime environment called Common Language Runtime (CLR). The source code of any program written in this environment could be viewed by using an intermediate language disassembler (ILDASM) or many commercially sold decompiler. We decided to provide the source code of the presentation layer to our clients. The business layer and the data access layer code were indiscriminate. The licensing code was written in the unmanaged code (native C++ code). The application programmer interface (API) was provided for the business layer and the data access layer for flexibility and extensibility.

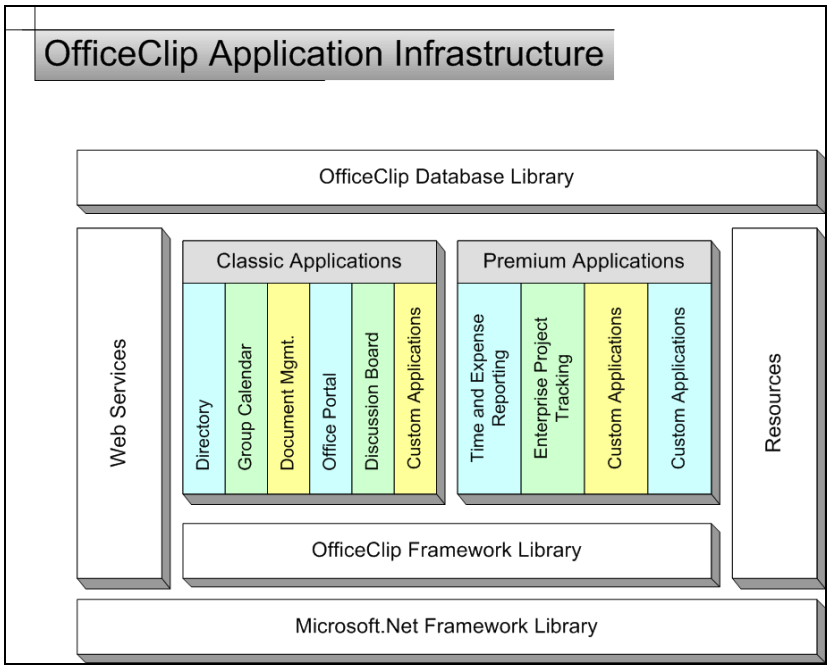

Figure 1: OfficeClip application infrastructure

Migration Strategy

Our initial migration approach was to write a translator to translate the existing code. We quickly abandoned this strategy, as this would not satisfy many of the goals we had set in the previous section.

The migration effort started with defining a new architecture of *OfficeClip*. A three-tier architecture was designed to support the application. A resource layer was added to isolate all of the resources required by the application, and a services layer was added to put all the clients and servers of web services. The entire migration process was divided into six parts:

1. Separation of the User Interface code while keeping the same user interface layout. This meant separation of the html from the asp code in the program. Because the layout was not changed, this was a relatively simple thing to do.

2. Separation of the embedded SQL commands from the code and the creation of a stored procedure for them.

3. Identification of common functions from the existing system and placing them in a common utility library.

4. Creation of the data access library to provide encapsulated access to the stored procedure.

5. Identification of all the business structures to create a business layer façade.

6. Identification of all the configuration structures with representation of them using XML schema.

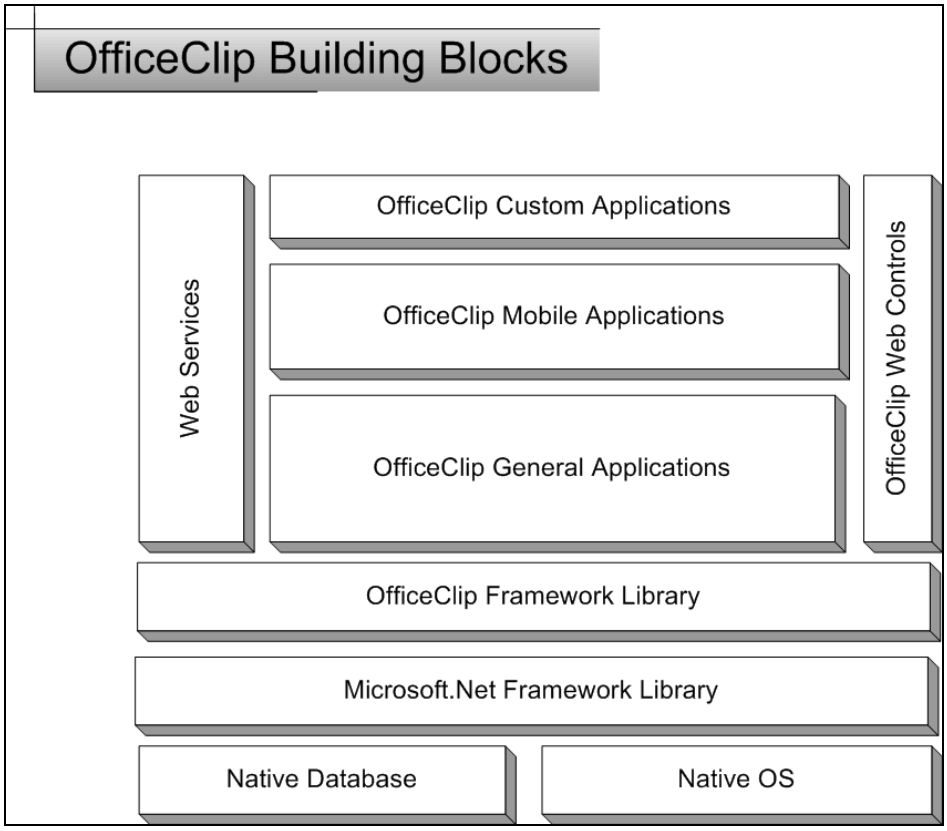

Figure 2: OfficeClip Building Blocks

OfficeClip Architecture

The three main layers of the *OfficeClip* architecture are the presentation layer, business layer and the data access layer. The user elements are coded in the .aspx files, which only have the html tags, framework control tags and the user-defined tags (for user controls). The business layer consists of business façade to access the information from the data layer. The data access layer contains the libraries to access the database.

OfficeClip uses web services as a primary vehicle to communicate with the outside world. A services layer encapsulates all the web services created and consumed by *OfficeClip*.

OfficeClip resources are composed of reports, images, localization resources and xml files. These are encapsulated in a different layer called the resources layer. *OfficeClip* users will be able to modify these resources to change the look, feel and functionality of the product.

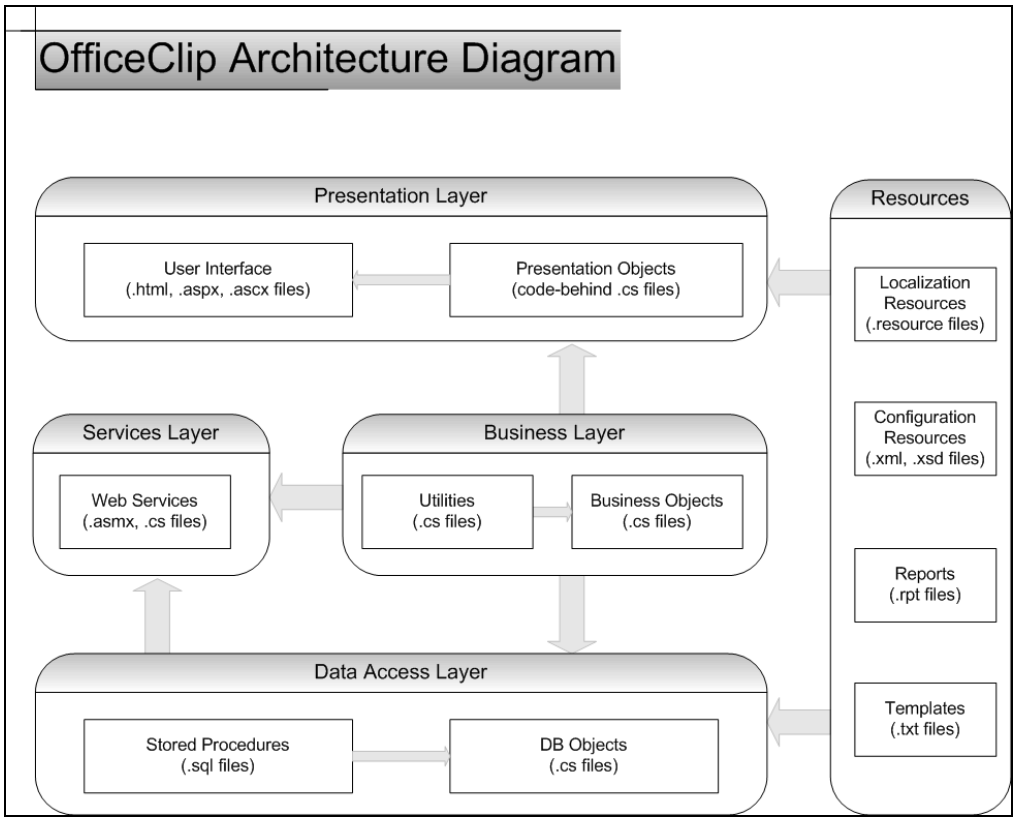

Figure 3: OfficeClip Architecture Layers

The interaction between various layers can be explained by the use of a UML sequence diagram (see Figure 4). When the user clicks to view all of the reminders in a list, a user event is generated that uses a delegate to service the event. It first instantiates a ReminderListObject from the business layer, and then populates this object by making a call to the data layer. Finally, the business object is removed from the memory and the delegate instantiates a new object to show the reminder list on the screen.

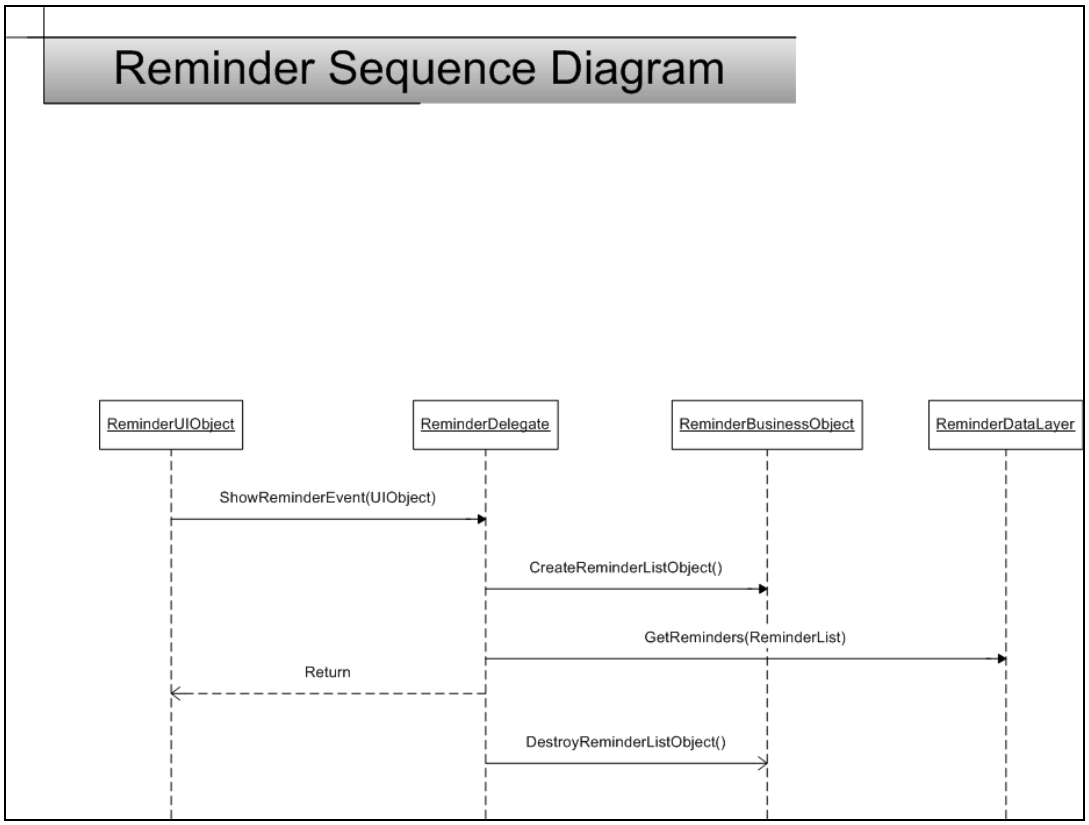

Figure 4: Reminder Sequence Diagram

Customization

The challenge any software product faces is to allow its user to customize it to the greatest extent possible. This exists even more so in application portals. At one end, we find shrink-wrapped applications that are inexpensive but difficult to customize. At the other end we find open systems' building blocks, where any kind of customization is possible but at a high cost. The Microsoft.Net environment has made the customization work a little easier by providing a rich variety of XML-based API, user controls and web services. There are three *OfficeClip* customization levels:

1. **XML- based Configuration Files (Level 1)**
   Most of the application parameters are configured in an XML file. These files are read at runtime and applications are configured according to the entries in these files. Some of the examples of the use of XML files are:

   - Controlling *OfficeClip* applications.
   - Controlling *OfficeClip* themes.
   - Setting up new mobile phones and pagers.
   - Creation of new templates for sending e-mails.

There are a total of about 30 XML configuration files that can be used to fine-tune the suite of applications in *OfficeClip*. These files can be modified by system administrators who will only need a minimal amount of knowledge on how to edit XML files. User interfaces to manipulate some of the common XML files are provided in the *OfficeClip* Enterprise Manager (OEM), which is a WhatYouSeeIsWhatYouGet (WYSIWYG) program to configure *OfficeClip*.

2. **Developer API (Level 2)**
   The business layer and data layer of all *OfficeClip* applications are open to developers by the use of an application programmer interface (API). The object-oriented framework of the Microsoft.Net environment provides leverage to inherit from the *OfficeClip* classes to modify them or to take advantage of them to design new applications.

3. **Source Code (Level 3)**
   The challenge facing users today is that they cannot yet modify the look and feel of a product by the using the vendor-supplied API. Vendors often use their own branded look and feel, which may not be suitable for the corporate environment. At other times, changing the presentation layer is a revenue source for the vendor, as the source code is not supplied. *OfficeClip* addresses these concerns by providing the source code of the presentation layer in the developer toolkit.

   Another challenge facing the software manufacturer and the users is that heavy customization often impedes the ability of upgrading the software. In many instances, the upgrade may overwrite the file, which is modified by the user. *OfficeClip* takes an approach which keeps the users' changes separated from the *OfficeClip* core files. All user changes are made to separate directory structures. A dictionary maps the user file with the existing files in *OfficeClip*. At loading time, the loader loads the dictionary and creates the map of the new resources in the memory. Thereafter, every access to the customized resources can be automatically found and directed to the appropriate resource. This frees the users to make changes to any *OfficeClip* program file or resource as they see fit, and also allows *OfficeClip* to provide patches and subsequent releases to the customer base without the fear of overwriting and changing files.

Localization

One of the migration goals was to make the software to work in various languages. This could not be done very efficiently in the current version of *OfficeClip* due to the cost and time to implement and maintain such an infrastructure. We wanted an infrastructure that would not be very intrusive to the existing code and could also be enhanced by adding new languages without compiling the code.

Microsoft.Net provides an environment for localization using satellite assemblies. These assemblies are created from the XML-based resource files and can be upgraded in the same way as other resources. In order to use this infrastructure, we simply changed all of the strings in our programs to a more generic name and created mapping files (one for each language).

The framework also provided capabilities to assign language to individual UI threads, so the same application could be run with different languages in different browsers.

One of the challenges that we faced while doing localization was that *OfficeClip* applications had about 400 images totaling almost 6MB of data. While these could be put into resource files and localized, we did not see that as the best option in consideration of memory space and efficiency. We implemented a mechanism similar to the satellite assembly and kept images in various local directories. At run time, *OfficeClip* would go and extract the appropriate image required to display on the screen. We implemented a fallback mechanism similar to the satellite assemblies in localization. If the specific image was not available for the culture specific language (e.g. fr-CA i.e. French for Canada), then it would fallback to a less-specific culture (e.g. fr i.e. French). If the less specific culture is not available, it would fallback to the *OfficeClip* default local culture (which is English US).

Mobile Access

The use of the Mobile API in the Microsoft.Net framework made it easy to create a mobile presentation layer. The other layers did not need any modifications. The Mobile API in the framework includes definitions for various devices, which takes the customization work off of the developer's shoulder. The initial mobile interface (a total of 10 screens) took about seven man-days to complete.

Results

At the conclusion of the project, we were able satisfy all of the migration goals we defined. In addition, the following are some interesting observations from our experience:

1. The code size increased by 50 % (450,000); however, the maintainable code decreased by about 50 %. This is because the application code templates were used to generate code in various layers.
2. The entire migration took about 1.5 man years, which should be considered to be a small number, taking into account the redesign of the entire code base.
3. Documentation of the code became much easier by following the disciplined approach of the Microsoft.Net framework and the use of the supplied tools. The resulting documentation could be created in XML, which could be easily translated into any format.

4. One of the major advantages gained from this effort was the code reusability. Heavy use of web controls made it possible to reduce development time on the *OfficeClip* applications.
5. Deployment became simpler, as the only outstanding item is whether the Microsoft.Net runtime is available in the target system. The installation and upgrade could easily be done by copying files to appropriate directories. There would be no need for installing and configuring DLLs.