SYS-CON

WHIDBEY: LOOKING AT MICROSOFT'S IMPRESSIVE OFFERING p.44

.NET

Presented by
.NET DEVELOPER'S JOURNAL
THE WORLD'S LEADING .NET RESOURCE
2004 Volume 2 Issue 12

Introducing SharePoint

WEB SERVICES WS

Integrate & extend your portal infrastructure

DotGNU Developers and Users Have New Portal
Mono holds 2nd summit

Master Pages in ASP.NET 2.0
Simplify reuse and maintenance

Using MVS2005 with VS.NET
Better use of server capacity

Extend Your Wings and Start the Migration

Moving a mature C++ CORBA application to .NET

BY PIPER KEAIRNES

.NET developers appreciate the ease of use offered by Visual Studio and the rich range of services from which to build enterprise applications. However, if you already have a successful product with an installed customer base, how can you take advantage of .NET? Do you need to start from scratch, or does it make sense to migrate a mature C++ CORBA application to the .NET framework? My team at VCG realized a number of benefits from tackling just such a project. In this article, I share our experience and show you how it all turned out.

The Challenge

Our flagship product, StaffSuite, was originally intended to run in a client/server configuration on Windows and UNIX operating systems with a back-end Oracle database. StaffSuite had been built using C++ and CORBA. The 6-year-old code base required a number of third-party libraries, which added cost and complexity. The StaffSuite architecture had grown organically, making it difficult to troubleshoot customer issues and time-consuming to respond to changing market demands.

On the business side, our sales force noted that our customers were standardizing on the Microsoft Windows operating environment. Prospects were no longer interested in mixed environment deployments. Some prospects and customers were interested in having the ability to extend our product to offer custom interfaces.

The Plan

To get better control over development and to increase the business value, the company agreed on the following goals:

- Upgrade to the Microsoft .NET framework to support a Web services architecture
- Reduce dependence on expensive third-party runtime libraries
- Deliver a more robust production environment with application failover capabilities and increased scalability
- Reduce application complexity to minimize the maintenance burden
- Simplify and improve the productivity of the development environment
- Develop a caching strategy for improved performance

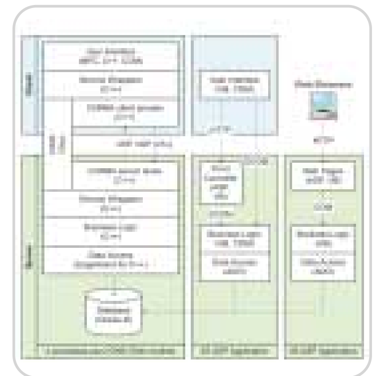
To meet these goals, we needed to make significant changes to the application at both an architectural and a coding level.

Simplifying the Architecture

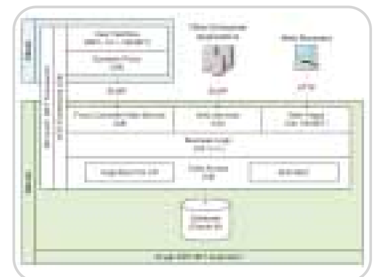
The original application architecture was heavily layered, with each layer performing a very narrow purpose. Figure 1 shows a simplified view of this layering. All totaled, adding one new service method required manual modification of at least nine source files as well as generation of the new CORBA client proxy and server stub classes. We wanted our developers to spend more time implementing functionality changes and less time marshalling client/server calls. We also wanted to eliminate the additional steps required to maintain IDL and generate CORBA client-server stubs.

The client and server architectures utilized a wide range of technologies including C++, VB, MFC, COM, DCOM, CORBA/IIOP, ASP, and ADO. Our goal was to reduce the variety and focus instead on a single technology platform.

The deployment environment was complicated by multiple network protocols, many separate executable processes, ASP applications, COM object registrations, and large numbers of DLLs. We strove to simplify the process model and reduce the overall file counts.



f1 **Figure 1:** *StaffSuite architecture*



f2 **Figure 2:** *The new StaffSuite architecture*

In contrast, our new version, StaffSuite 4.0, is built on a service-oriented architecture with clients connecting to the .NET server application via SOAP. StaffSuite can also support browser-based clients through HTTP. Within the StaffSuite server, business logic is written in Managed C++ and C#, and data access classes are automatically generated by Progress Software's EdgeXtend for C#, an O-R mapping and caching product. Figure 2 shows the streamlined StaffSuite architecture.

HOME

CLIENT

SERVER

EVERYWHERE

AUTHOR BIO
Piper Keairnes is the lead architect at VCG, Inc (www.vcgsoftware.com), a solutions provider to the staffing/recruiting industry. He has been leading the development of commercial enterprise applications for 8 years, spending most of that time in Java/J2EE and .NET.

Microsoft .NET provides a single technology platform that allows us to simplify our product architecture and deployment environment by doing the following:

- Replacing CORBA and DCOM with Web services and the ASP.NET runtime
- Utilizing Reflection and Dynamic Proxies to eliminate multiple architectural layers
- Eliminating our use of COM
- Tightly integrating C# and legacy C++ and VB code within the same application
- Hosting all our server logic within a single server process

We eliminated CORBA, DCOM, and several architectural layers by replacing the client/server communication stack with a client-side Dynamic Proxy, simple C# interface definitions, and a single Web service method. Now when developers add or change a service method, they modify only two files: the C# interface file and the server-side C# or C++ class that implements that interface. Figure 3 shows a class diagram of this mechanism and highlights in blue an example of the two types of classes that need to be modified.

Because this pattern uses reflection to delegate the service requests to service classes, the top layer of legacy C++ code had to be converted to Managed Extensions for C++. All new service classes are being written in C#, and these new classes collaborate with the legacy C++ classes through the ServiceFactory and service interfaces. For server-to-server calls, the ServiceFactory optimizes performance by returning a reference to a concrete service class (e.g., EmployeeService) rather than the DynamicProxy that is used for client-to-server calls.

This pattern uses a fair amount of reflection, which might cause some people to question performance. However, the actual cost of reflection is negligible compared to the object serialization and network-database I/O involved in these coarse-grained client/server calls.

Our application did not make heavy use of COM components, so we took some time to eliminate that technology from our architecture. We were able to replace several COM libraries by switching to comparable classes in the .NET Framework or by using open-source .NET projects such as SharpZipLib. Finally, we upgraded some third-party libraries to their newer .NET versions.

Last, because of the fact that .NET supports multiple programming languages, we were able to integrate our server-side C#, C++, and VB code into a single ASP.NET application domain. The server was previously deployed as two ASP applications, four server processes, and a Java-based Orbix runtime. Now all of these components run in a single application directory within IIS. This architectural change dramatically simplified the way we deploy and administer the server.

Leveraging the Code

Data access is extremely important for StaffSuite, which offers users a wide range of functionality including sales, operations, payroll, billing, and government reporting. The application object model consists of hundreds of data classes that map to 471 data tables and a number of views. Nearly half of the tables exist to support configurable lists and workflows that result from this complex data model.

The standard .NET interface for data access is ADO, which proved to be the best fit for one of the StaffSuite modules for client-side functionality. However, this module was based on a simple data model that translated easily to ADO. Previous versions of StaffSuite had used Progress Software's EdgeXtend in place of C++ to generate the data access logic. With the complex object model on the server side, our team did not want to face the

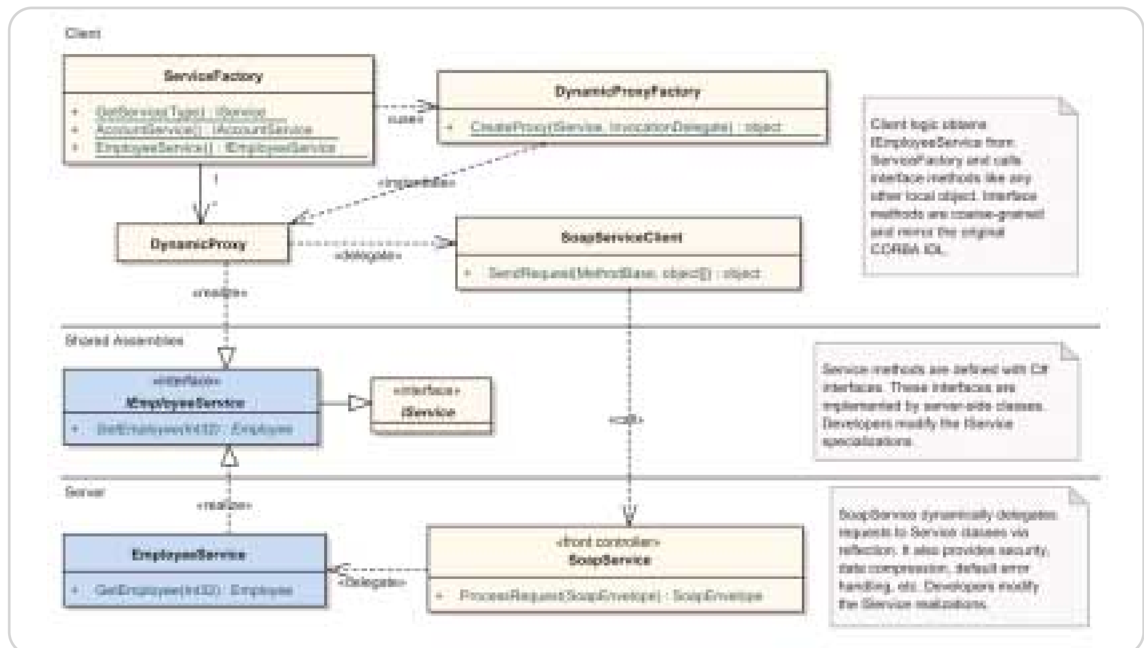


Figure 2: A class diagram

HOME

CLIENT

SERVER

EVERYWHERE

daunting task of converting all data access to ADO.

When we learned that Progress was developing a C# version of EdgeXtend, the development team was enthusiastic about joining the beta program. We felt that retaining this object-oriented data access paradigm in the business logic layer would offer greater ease of use and higher productivity.

Using EdgeXtend to generate code also relieved us from concerns over simple syntax errors or misspellings. We were able to use C++ and C# compilers to detect code that was impacted by a schema change, which is a more reliable approach than manually reviewing SQL statements. Just as important, the team wanted EdgeXtend's ability to cache unpredictably changing data to tune application performance and scalability. Neither ADO nor .NET have provisions for automatically caching dynamic data.

EdgeXtend's generated data classes fully implement all database access and object model constraints. Just to give you an idea of the difference between the amount of effort required to provide data persistence using ADO and the EdgeXtend approach, consider a simple relationship in which two database tables (WIDGET and PRODUCT_LINE) have a one-to-many relationship. The foreign key (ProdID) is on the WIDGET table (see Table 1).

Listing 1 shows how the same operation would look invoking EdgeXtend's generated O-R mapping code. Notice how the EdgeXtend solution uses no SQL or database-specific code. The relational data is treated the same as any ordinary .NET object. Obtaining information in related tables becomes as simple as accessing an object attribute (prod.widgets).

In ADO.NET we use strings to describe the data we want from DbCommands, DataRows, and DataReaders. This results in a class of issues that can only be discovered at runtime with thorough unit testing of your data access logic. In contrast, EdgeXtend provides a less error-prone data access environment because data is accessed in the form of type-safe objects, properties, and relationships. If you misspell the property "widget.ProductLine.Name," the C# compiler will let you know. But with ADO.NET SQL strings, you won't know until runtime that you had fat-fingered the syntax for your inner join. Besides the obvious productivity improvement of not having to hand-craft SQL statements, you also improve quality and developer productivity by catching some errors at compile time.

EdgeXtend automatically caches dynamic data. After the first pass through the code in Listing 2, the ProductLine and Widget objects remain in memory until they are garbage collected by the .NET runtime. During a second pass through Listing 2, if these objects have not yet been collected, EdgeXtend will return the same objects without needing to hit the database again. That's a wonderfully transparent performance improvement. But what happens when you scale your application by adding another server? The answer is simple: just change your configuration file to tell EdgeXtend to automatically synchronize these distributed caches.

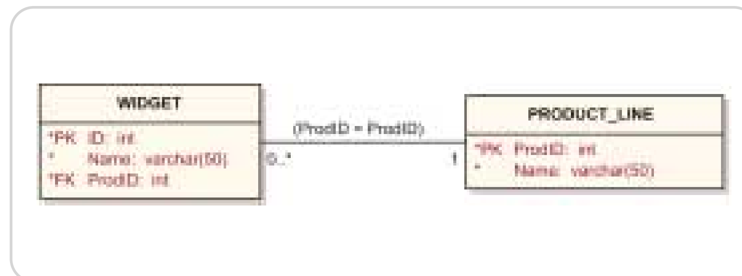


Table 1: *Relational data tables*

Development Benefits from Using EdgeXtend

Choosing EdgeXtend for data access allows us to simplify our development process and to provide more features in less time. It helps reduce application complexity, improves developer morale, and lowers overall costs.

- Reuse of the application's existing object model and business logic dramatically shortens development time. Through the use of model-driven tools, the team ported 471 data classes from C++ to managed C# in 3 days!
- Iterative code generation allows for agile development practices, which improve team morale. Rather than having to get object model and database mappings exactly right the first time, we could concentrate on one area, tune it, and then move to the next.
- EdgeXtend's optimized generated code reflects years of research and development, and substantially reduces debugging and testing requirements.
- We were able to use new architectural patterns. With a C# data access layer, VCG could use .NET Reflection. This eliminates or dramatically simplifies several architectural layers and subsystems. For example, StaffSuite's original C++ architecture requires a complex set of function templates to dispatch requests and provides a consistent structure for updating the large number of persistent classes. In contrast, the new architecture provides simple reflection-based dispatchers that make it easy for all new server-side logic to be written in C#, all the while maintaining transparent interoperability with existing C++ business logic. .NET Reflection also enabled a large reduction in source code.
- The move from C++ to C# reduces the build time for StaffSuite's large data access layer from 90 minutes down to 20 seconds.

Benefits of Moving to .NET

The move to .NET resulted in productivity, quality, and cost benefits, allowing us to:

- Address market demand for customizable browser-based applications, set the stage for new Web services, and compete more effectively.

- Eliminate several third-party software components by leveraging the .NET framework to lower the cost of deployment and simplify application architecture.
- Reduce application code by 30% and the number of classes by 50%, simplifying development and significantly decreasing debugging and maintenance efforts. A significant part of code reduction lay in removing third-party components such as CORBA. This helped in avoiding CORBA proxy classes and affiliated layers that mirrored the CORBA interfaces and also eliminated tens of thousands of lines of code.
- Uncover existing C++ bugs with the C# language complier for improved code quality.
- Reduce total application build times from 8 hours to 15 minutes.
- Simplify deployment while improving manageability for application administrators.

Benefits to Our Customers

Ultimately, we must gauge the success of this project in the market. Customers and prospects have reacted positively to our new focus on .NET because it generally mirrors their own technology direction. Salespeople have found it an advantage to offer a .NET solution over a CORBA or Java solution.

In addition to the new business functionality in the 4.0 release, StaffSuite customers will benefit from all the technology upgrades in some important ways:

- ***Simpler installation:*** Version 4.0 is faster and easier to install than previous releases, with fewer components creating fewer potential installation problems. StaffSuite users must upgrade their software each year because they must comply with new tax rates, tax codes, government regulations, and changed government reporting file formats. A streamlined upgrade process benefits existing customers.
- ***Better performance:*** StaffSuite 4.0 is noticeably faster than previous versions because of improved data caching and performance tuning.
- ***Higher scalability:*** Through the increased scalability provided by distributed caching, the 4.0 release will benefit VCG's largest customers and help VCG appeal to even larger prospects.
- ***Greater reliability:*** Through the addition of application failover, the upgraded architecture is more "bullet-proof" than previous versions, minimizing support calls.
- ***Easy integration with other applications:*** With Version 4.0, VCG's larger enterprise customers will be able to integrate StaffSuite with other systems in their environment at the application layer rather than the database layer. For example, new data (e.g., a resume or timesheet) that comes from another source can be validated and synchronized by StaffSuite's business logic instead of being added directly to the database.

Conclusion

StaffSuite 4.0 went through a very successful round of beta testing in August and September, and the product was released in early October. Even with all the architectural changes, component upgrades, deployment changes, and functionality improvements, Version 4.0 is turning out to be the highest quality release yet.

During beta testing, we enhanced application performance by focusing on EdgeXtend data caching. Cache clustering and failover features will be important to VCG clients who need scalability and high availability. As StaffSuite evolves, the database independence of the EdgeXtend tool will also allow us to meet customer demands for SQL Server support without requiring significant code changes.

We've found the .NET framework to be an excellent platform for development and deployment. The simplified architecture and reduced code base delivered significant productivity and quality improvements. If you weigh the benefits against the risks, you may decide it is worth the effort to migrate a mature application to .NET. ◉

Listing 1. Display all widgets of the product line using ADO.NET

```
using (SqlConnection myConnection = new
SqlConnection(connectionString)) {

     string sSQLquery = "select W.ID, W.Name,
PL.Name"
          + " from Widget W inner join Product_Line
PL"
          + " on (W.ProdID = PL.ProdID)"
          + " where PL.Name = 'BestSeller' ";

     SqlDataAdapter theAdapter = new
SqlDataAdapter(sSQLquery, sConn);
     DataSet widgetData = new DataSet();
     theAdapter.Fill(widgetData);

     foreach (DataTable table in widgetData.Tables)
     {
          foreach (DataRow row in table.Rows)
          {
               Console.WriteLine("Widget ID: " +
row["W.ID"].ToString());
               Console.WriteLine("Name: " +
row["W.Name"].ToString());
               Console.WriteLine("Product line: " +
row["PL.Name"].ToString());
          }
     }
}
```

Listing 2. Display all widgets of the product line using O-R mapping solution

```
// Instantiate the correct product line. Perform db
call under the covers
IList products =
ProductLineFactory.FindByName("BestSeller");
foreach (Product prod in products)
{
     // Get all widgets associated with the product
line
     IList widgets = prod.Widgets;
     foreach (Widget widget in widgets)
          {
               Console.WriteLine("Widget ID: " +
widget.ID);
               Console.WriteLine("Name: " +
widget.Name);
               Console.WriteLine("Product line: " +
widget.ProductLine.Name);
          }
}
```