DESIGN FOR EVOLUTION

WHITE PAPER

By Chris Armbruster

ABSTRACT

With rapidly evolving business needs and Internet technologies, competitive advantages will be gained by those who anticipate and prepare for the future. Although Internet applications are often backward compatible today, they are seldom forward compatible. An application that exhibits a component-based architecture, extensible interfaces, and appropriate use of abstraction will lend itself to forward compatibility, providing an adaptable tool for its users and a foundation for its developers to move ahead on the evolution curve. "Design for Evolution" is an exploration of forward compatible design for Internet applications. It includes a case study of PowerQuery, an evolving suite of Internet applications that supports material planning and provisioning for the Columbus GPC, using ODBC, Server-side Scripting, COM, ActiveX and DHTML technologies.

INTRODUCTION

At a press conference in San Francisco, Microsoft Chairman Bill Gates remarks that the "… Internet is a passing fancy." This was in October of 1995. Two months later, Bill Gates recants saying "…not only is the Internet not a passing fancy, but we are going to move aggressively to incorporate Internet access into all of our product lines." At that time Microsoft had ZERO revenue from Internet or web-related software. In December of 1996, just 1 year later, Microsoft closed its books with about US$4 Billion in web-related revenues.

How did Microsoft do it? How did they change their strategy and technology in what seemed to be over night? While many would argue that it was their sheer size and abundant resources, they could not have pulled it off without an architecture that was designed to evolve. For if it failed to evolve, it could have threatened their very existence.

Business and technology: two dynamics that command every developer's attention. Changes in business processes and strategies are inevitable and sometimes radical. Technology is evolving at an increasing pace. We are in new age…digital, genetic, information-based and wireless…an age where everything is changing. Since developers work in this same environment, often building tools to facilitate it, they will either design for evolution or extinction.

These dynamics should not be feared – they should be prepared for. Any developer can do this by following three fundamental design principles: extensibility, abstraction and componentization. This paper explores the dynamics that are driving these principles, as well as the principles themselves and how they enable applications to evolve. It emphasizes design principles and while it may use specific technologies as examples, it is not intended to make an argument for or against any single technology. The technologies adopted to support the principles discussed in this paper are left to the discretion of the developer.

EVOLUTION

Why do some applications seem to linger around forever while others come and go? How do certain developers produce one successful application after another while the products of other developers fail before they even get to market? What are the factors that determine the success of a software product? In a large way, the deciding success factor for applications is the ability to evolve. Business and technology are changing so fast that if an application cannot evolve with them, it will become extinct. The ability to survive depends on adaptability and natural selection.

The business climate is changing in many dimensions, making it increasingly harder to keep up. The industrial revolution is over and we find our selves in an age of information…many companies now sell information and tools for using it. The world has become digital and virtual…cars are ordered on the Internet without ever being driven…cameras use memory rather than film. We live in a culture of immediacy…6 to 8 weeks delivery is no longer acceptable, it has to be over night…9AM to 7PM no longer works at a grocery store, they must remain open 24 hours. It's chaos…companies make and break deals…everyone seems to be in a merger…or split-up…corporations are bought and sold like candy. Applications can no longer be designed for a given environment. They have to be designed for all possible environments.

When I first started working with information technology back in college, it was very different and by today's standards crude. Since then, technology has evolved at such a rapid pace that I strongly doubt Captain Picard will be reading *bound* classic literature 400 years in the future. It will more than likely be a digital representation. Internet technology alone has changed so much over the past few years. There are new media types, new protocols, faster access, more people connected and new ways of building applications. We have also seen an evolution in the World Wide Web…from static web sites to CGI to server side scripting to distributed component technologies. The growth in technology is not limited to information, it can be seen everywhere…genetics, healthcare, space, etceteras. New technologies often mean better ways of doing things. Surviving applications readily adopt the technologies that offer advantages.

Natural selection, in terms of Internet applications, is the phenomenon by which users and developers naturally choose that which works best. Since so much continues to change, the selection process is ongoing. What worked best yesterday won't necessarily work best today and today's solution might not be tomorrow's. For developers, natural selection means adopting the development environment that works best. It is easiest to use, most reliable, fastest to deploy, leverages existing knowledge, and is best supported. Different developers will weight these differently. The long-term successful developer will adopt development tools that allow him to build applications that evolve quickly, cost-effectively and reliably. The tools developers use will play a big role in which technologies are adopted. It isn't necessarily what is most often chosen by the developer, however, it also depends on the user.

The users will naturally choose what works best for them. Even if 99% of developers adopt technology A while the remaining 1% adopt technology B…if technology B works better for the user or offers the user a competitive advantage, then this will be the technology that is adopted. Developers must pay attention to what the user needs. Users will choose what works best for them, not what works best for developers. Users that change business rules frequently need developers who use technologies that allow business rules to change easily and rapidly. Users whose mission depends on real-time information need developers who use technologies optimized for delivering real-time solutions. Application survival depends on the natural selection of both the user and the developer.

Here are a few questions that every Internet developer should be asking about their application:

What happens to my application if the business rules change?

What happens if we enter a different market place?

How much additional work will be required to incorporate new technologies like speech recognition, natural language query and hand writing recognition?

What about Internet devices that don't look like computers such as Web TV, PDA's and cellular telephones?

What if the technology for storing data changes?

What separates the successful developers from the ones that come and go are the questions they ask themselves. Designing solutions that provide good answers to the questions above is what designing for evolution is all about. Since technology and business processes cannot be accurately predicted, developers that design for evolution in general will be better prepared for any change.

In this section we discussed how the environment is changing and the importance behind designing applications that can evolve with it. The incredible rate of change should not discourage the developer, however. By following a few fundamental principles, applications can be designed to evolve. Concentrating on design principles rather than technologies, results in principle-centered, rather than technology-centered, design. Principle-centered design does not bind itself to technology or business rules. If you base your design on the right principles, then the technologies chosen become less important. Three fundamental design principles lend themselves to forward compatibility and evolution: extensibility, abstraction and componentization. Use of these prinsiples results in solutions that provide encouraging answers to questions about change. The balance of this of this paper is an exploration of these principles and a case study that leverages them.

EXTENSIBILITY

Extensible means having the capability to be extended. If an application is extensible, then it is not limited to the methods, protocols, content, etc. that were considered at design time – it can be extended to handle new features. It can evolve gracefully. In a client/server environment like the Internet, you will, at any given time, find applications in various states of evolution. That is, technology "cuts" aren't made all at once because everything doesn't change at once. It may start with just a few clients and a single server. This means that applications must be backward compatible as well as forward compatible. It is easy to make a new application support old technology, but making an old application support new technology can be tricky. The key is to give applications the ability to negotiate usage-level details and to design them to degrade gracefully with new technology rather than fail miserably. In this section, we will look at several ways of doing this.

In the early 90's, work began on the now popular Hypertext Transfer Protocol, better known as HTTP, by the W3C. HTTP was designed to be a generic, object-oriented network protocol for retrieving resources located on a network (see Figure 1). Since the original HTTP developers could not possibly know of all the different types of resources that might be retrieved using their protocol, they needed an approach that would not bind them to what they could think of at that time. They needed something that didn't care about the type of resource being retrieved. The result was a protocol, extensible in at least three dimensions, that has handled Internet evolution gracefully and effectively. By employing the same principles that lie behind the extensible design of HTTP, developers can design applications that extend just as well. The balance of this section will use the HTTP protocol as an example to help explain design considerations for extensibility.

Figure 1

HTTP was originally designed to support three key methods or procedures (GET, POST and HEAD). Basically these are operations to be performed on the Internet resource (or URL) to which they are applied. For example,

GET HTTP://WCS.CB.LUCENT.COM/EVOLUTION/DEFAULT.HTM

applies the GET request method to the default.htm document located on the server referenced in the URL. The GET method tells the server to retrieve the document. If GET was replaced by HEAD, the server would return headers, but no document. HTTP is extensible in terms of the request method; that is, new request methods may be used. All that is required is that the client and server know how to handle it – if the server doesn't, it won't crash, it will just return "method not supported." This means that new clients can be deployed that use new methods, but they can still work with old servers. If they get a message from the server stating that the method is not supported, then they should resort to an older method, which the server is more likely to support. When a method is not supported, clients shouldn't fail, but rather degrade gracefully. The same principle, method extensibility, can be added to virtually any Internet application.

Similarly, a developer can design his methods so that the arguments for them may be extended. Methods usually have a fixed number of possible arguments with a specific order. For example:

PrintAddress(Name,Address, City, State, Zip)

But what would happen if you needed to add a company name to the address. You could either force it to fit in an existing argument or you could recreate the method to handle it as its own argument. A more extensible approach is to design a method that doesn't care about the number of arguments or order. For example,

PrintAddress(Name=*name*, Address=*address*, City=*city*, State=*state*, Zip=*zip*)

This approach includes the argument name with the argument so the order is unimportant. What happens if you add a new argument? All you have to do is append it to the list. If the process doesn't recognize it, it can just ignore it. This way, older implementations of this method won't crash when being accessed by newer resources.

HTTP uses headers to communicate meta-information. The protocol is not restricted to the headers that were conceived at design time. The specification only requires that the headers be formatted as attribute and value pairs with a colon between them and a new line for each pair. Basically this means that you can pass whatever meta-information you want. All you have to do is follow the formatting rules. For example, let's suppose that

EVOLUTION : YES

is an HTTP 9.0 (by the way, there is no HTTP 9.0 yet) header telling the server to evolve. This doesn't have much meaning to HTTP 1.0 proxy servers because the header doesn't exist yet. Nevertheless, an HTTP 1.0 proxy server may see the header in disparate environment. If the proxy server didn't have a way to deal with the header, the request might not make it to the server. Instead, the proxy server passes this header on, only caring about format (attribute : value). This facilitates evolution by allowing new technology to be supported in an old technology environment.

The use of attribute and value pairs is an object oriented approach that allows developers to design applications at the object level and not have to worry about every possible attribute that could ever be used; new attributes evolve without a scrap and redesign of the application. Let's suppose you were designing an electronic catalog for a factory that manufactures cell sites. The catalog may have product attributes like power, frequency and a digital image. If the supporting database table was designed so that each of these attributes was a field, then what would happen when a new product had a fourth attribute, like color? You would have to add a field to the table. Eventually your table could have more fields then it did records. A better approach is to represent all attributes with just two fields, attribute name and value (see Figure 2). This way, as new attributes evolve, the application can be extended to handle them with no need for a redesign. The same goes for the rest of the application that uses the information. If it treats everything as an attribute and value pair rather than individual attributes, it won't care when new attributes are added.

Part #	Attr	Value	Type
123	POWER	DC	STRING
123	FREQ	3.4	FLOAT
123	IMAGE	BLOB Data	IMAGE/GIF

Instead of

Part #	POWER	FREQ	IMAGE
123	DC	3.4	BLOB Data

Figure 2

New types of content continue to appear on the Internet. Five years ago, there was no such thing as DHTML. Now everyone is talking about it. How do old applications exist in a world with new content. The answer is negotiation. By the use of message headers, HTTP clients can advertise to a server the types of content they support. A smart server will try to return content the client can handle. For example, an image server can contain multiple formats for each image and return only the format that the client supports. This allows a new format to be adopted without cutting off service to old clients. This also allows HTTP to leverage other Internet standards like MIME.

Similarly, applications can be developed to negotiate. This doesn't just mean content type. As thin clients become more prevalent, they may not be able to handle large content due to memory constraints. Size is

something that can be negotiated. Data types may be negotiated as well. Suppose an application designed to perform data collection on the Internet was designed at an earlier time than the resources that it communicates with. What happens when a Binary Large Object is sent to a resource that doesn't support such a data type? It might crash or it might fail gracefully. The better answer is to negotiate the content prior to exchanging it. This way something that the resource can use may be sent.

In this section , we looked at design techniques that allow applications to be extended. The HTTP protocol was used as a working example due to its great success in surviving the changes in Internet technology over the past few years. It's success, in large, is due to the extensible design principles on which it is based:, method extensibility, attribute and value pair usage and content negotiation. These same principles can render long-term successful results for developers that consider them at design time.

ABSTRACTION

To abstract means to withdraw, take away or separate. A system with an abstracted architecture isolates sub-components from each other and from the whole. This allows pieces of applications to be built independently of each other and evolve with greater ease. Consider a web server that collects user data, applies business rules, formats the results and then returns them to a browser. What happens if you change the business rules? It depends on whether or not the application abstracted the rules from the rest of the application. If they weren't abstracted, then most of the application will probably need to be redone. If, however, the rules were abstracted then all that needs to be done is to change the rules layer, a far simpler task than redoing the entire application. Abstracted architectures let you replace any layer without affecting the others. All that is required is that you follow the rules for interfacing with the other layers. Abstraction makes evolution easier because it allows layers of an application to evolve independently of each other—so whole applications don't need to be rebuilt, just the layers that are affected by the change. This section will explore several examples of abstraction in use to demonstrate its applications and benefits.

The Windows NT operating system abstracts the software from the hardware on which it is built to run. It does this by putting a layer between the operating system and the hardware. This is known as the hardware abstraction layer, or HAL (see Figure 3). The advantage of this is that if a new hardware platform comes along, all that needs to be done to make Windows NT compatible with it is to write a hardware abstraction layer specific to the new platform. The result is an operating system that currently works on four different platforms and is scalable from 1 to 256 processors. But, that's not all, the real advantage is that this positions Windows NT to evolve when changes in hardware technology occur without having to start from scratch. With the increase in embedded Internet technology, developers have more reasons than ever to abstract the software from hardware.

Figure 3

Similarly, the Java Virtual Machine (JVM) provides a layer of abstraction between the platform and the Java code that runs on the machine. This allows you to write a Java program once and run it on any platform. All that is required is that the platform has a JVM. This also gives developers a nice mechanism for hosting foreign applications.

Almost thirty years ago, the Department of Defense (DOD) funded an effort to connect some computers together. The project evolved from a handful of sites to the Internet as we know it today. How could one model be so scalable—from a few sites with limited applications to countless sites with endless applications? The answer is that it was explicitly designed to evolve. Out of the early work came the DOD Reference Model (see Figure 4). It is a model for the Internet that abstracts an application running on it into four layers: application, host-to-host, Internet and network access. The model specifies how each layer interfaces with the other. By doing this, any layer can be replaced and the model will continue to work providing that the layer is compliant with the interface specifications. This is why your web browser doesn't care whether the physical network is phone line, category 5 wire, coaxial cable or fiber optics. The network access is abstracted from the layer that the web browser runs on, the application layer. This is also why the HTTP could be implemented years after the Internet was launched. No changes were required in any of the other layers, so building a new protocol meant just writing a specification. By building applications that are based on layers of abstraction, the developer is designing for evolution.

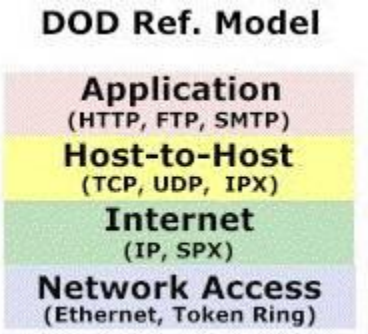

Figure 4

Today's client/server applications resemble their ancestors so little, they are now called n-tier applications. In this model, the division between the client and the server isn't as easily discernible, and often the server becomes a client itself. The new model abstracts and distributes tasks more evenly between the client and the

server and often includes other resources as well. Viewed from a purely functional standpoint, most systems will perform three main tasks: presentation of the user interface (UI), business logic and data services (see Figure 5). The n-tier architecture isolates or abstracts each major piece of functionality. The presentation is independent of business process rules, which in turn is separate from the data. The result is a system that allows you to change the business rules, something that is very common today, without having to change the user interface or the data service. What if a new UI technology was becoming popular, but not yet ubiquitous? This architecture will allow you to add additional UI's without having to redo the business rules or data services. This is becoming increasingly important as we see new UI technologies like speech recognition, PDA's and embedded devices evolve. It would also allow multiple UI's to coexist – making evolution very smooth. What if you wanted to change your database technology to another vendor? Would it require a complete rework? Not with this architecture, you can plug in any database that meets the interface specifications of your architecture. In today's data intensive environment where data exists in so many places and with new data sources becoming available all the time, binding yourself to the technology of any specific data source is likely to mean extinction. This architecture separates the data technology rather than binding to it. Developers who harness the n-tier architecture are rewarded with very flexible applications, applications that have few limits and the ability to evolve.

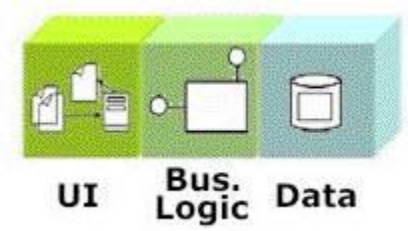

Figure 5

In exploring abstraction, an approach to isolating the components of an application from each other, we looked at hardware abstraction, a network reference model and the n-tier architecture. We discussed how separating the pieces of an application requires less overall work when business rules change, resulting in a solution that evolves rather than becoming extinct. Layers of abstraction were discussed along with the need for an interface specification between layers. The benefits of isolating business logic in a high change environment were studied as well. We also looked at the need to abstract data in data intensive environments. This discussion is not all-inclusive, but rather an introduction. The developer should try to understand what technologies might change and then abstract them from the application accordingly. Here are a few other items worth considering: content-type, protocols, interface documents and the operating system. The developer that invests the time in analysis and design up front greatly reduces maintenance costs and increases the application's ability to evolve.

COMPONENTIZATION

Componentization is the breaking up of monolithic applications into separate binary executables that connect with each other at run-time to form an application. Like abstraction, componentization separates an application into sub-components, but it does more. It offers a binary specification, a mechanism for upgrading and fosters dynamic linking. The component-based architecture is a very specific way of designing an abstracted solution. It is entirely possible to design applications that are abstracted in more ways than just by components. Componentization allows applications to have components easily replaced, making evolution on a component-

by-component basis possible. The principle also permits applications to be formed by simply gluing a bunch of components together with script or code. In this section we will look at the benefits of componentization and how such architecture is evolution friendly.

Traditional applications are usually made up of a single monolithic binary file (see Figure 6). After being compiled, the application doesn't change until a newer version is recompiled and distributed. Hence, these applications are static (and many times out of date) when they ship. Today, information technology is changing so quickly that it is no longer feasible for an application to behave this way.

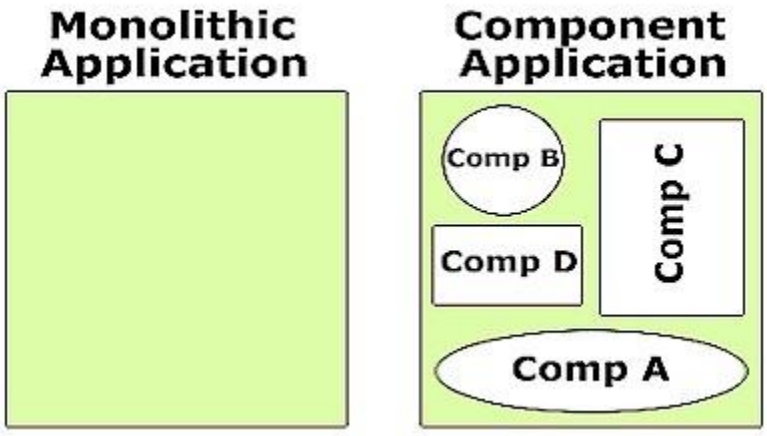

Figure 6

By breaking up the monolithic application into separate components, change is more appropriately addressed. Applications can stay up to date by merely upgrading components. With today's technology, this can often be done transparently and dynamically on a network.

The requirements for components are:

- Language independence
- Shippable in binary form
- Upgradeable without breaking old clients
- Transparent location and relocation on a network
- Dynamic linking

Application customization is one trait of componentization that lends it to evolution. Let's suppose an application was designed with an embedded Java scripting engine. What happens when you acquire another group of users, a group that prefers Visual Basic script? This could mean a bunch of disgruntled users. If the architecture treated the scripting engine as a component, however, it would just mean replacing it with a Visual Basic scripting engine component, something that can be purchased over the counter with no development necessary. What happens when someone wants to use Perl Script – the same thing!

Component-based architectures give developers the ability to isolate highly changeable aspects of applications, like business rules, into components. Imagine the work involved in modifying all of your applications when the rules change. When the business rules are coded into components, a single set can be used to service every application that depends on business logic. If the business rules change, all you have to do is change one component, rather than every application.

In today's changing environment, the ability to easily upgrade an application is a must. What if you needed to add new features to a component that initially only a few clients would benefit from? Wouldn't it be nice to upgrade the features of a component yet still have them behave the same to applications that don't take advantage of them? Component-based architectures provide a facility for this through the negotiation of interfaces – one for the old client and one for the new client. This allows multiple generations of applications to coexist – making evolution graceful.

The shrinking of the developer's time-to-market is making rapid application deployment more prevalent than ever. By designing a set of linked, reusable components, new solutions can be developed overnight by merely linking together a set of existing components with script or code. Imagine the time saved by not having to re-code business rules or data access. By developing to one of the popular component specifications, the developer may also purchase off the shelf components to incorporate applications. This would result in a library of components that could be reused whenever an application is being created or modified, shortening the development cycle.

Componentization makes implementing distributed applications easy. A distributed application divides tasks between two or more machines (see Figure 7). The client/server and n-tier models render distributed applications. Since a component-based architecture allows components to be transparently located and relocated, distributing them among different systems is entirely possible. It allows applications that were originally designed to be hosted on a single system (using older component technologies) to evolve into a distributed system by replacing the older components with remoting components. The ability to distribute components also facilitates architectures that centrally locate business process components on a single machine for all applications to access. Component-based architectures provide an effective solution for the evolving trend to distribute applications.

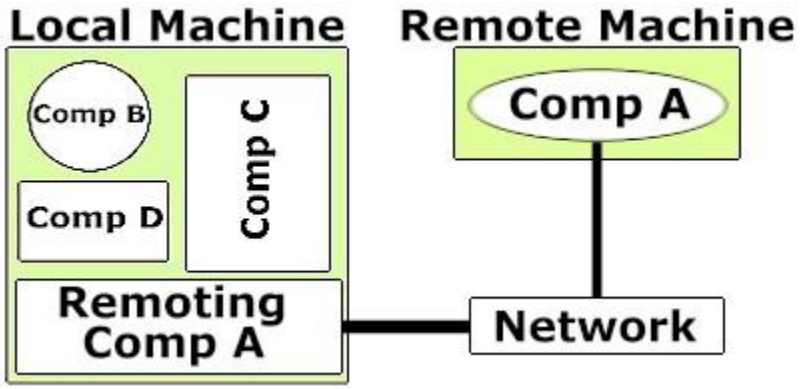

Figure 7

When adopting a component-based architecture, the developer has two choices. He can use an existing technology or he can create his own. First, we will look at leveraging an existing technology. By adopting an existing standard, the developer will be spared the cost and time associated with developing a standard. The

developer can further shorten the development cycle by using off the shelf components compliant with the existing standard. Hiring additional developers becomes easier because the chances of finding candidates with experience in a public technology is more likely than finding candidates with experience in a private one. Development environments are also available to those that design components compliant with an existing standard, making everything easier. Adopting an existing technology clearly has its benefits.

What technologies are available? Although scriplets and applets offer some of the same features of componentization, they aren't full implementations of the architecture. DCOM and CORBA are two competing specifications that provide a sound basis for architecting a solution. DCOM, or the Distributed Component Object Model , is a standard proposed by Microsoft. It currently runs on Windows platforms with implementations being tested on UNIX. DCOM, COM , COM+ and ActiveX are all very closely related specifications offered by Microsoft, some of which are subsets of the others. It is beyond the scope of this paper to delve further into these technologies. The Object Management Group (OMG) is a consortium of companies that have been developing a component communication standard as well. OMG is responsible for CORBA, or the Common Object Request Broker Architecture. Implementations of CORBA are available for various platforms, including Windows and UNIX. Although DCOM and CORBA both offer standards for inter-object communication, provide facilities to implement distributed applications, and encourage the separation of business logic from the presentation layer, the implementation of these standards is different and they should be considered compatible. The technology adopted is left to the discretion of the developer.

What if you wanted to brew your own specification? This is entirely feasible, but you would lose so much. You couldn't use off the shelf components or leverage development environments designed for existing standards. You would have trouble finding developers with experience in your proprietary specification. You would also have to invest the time and money to design and test your specification. So, why do it? Maybe DCOM or CORBA don't entirely meet your needs. Maybe you want something simpler. Maybe you want to sell tools for your new specification. Although there are many factors that encourage the use of an existing standard, the ambitious developer is not precluded from creating his own.

In this section, we studied componentization: a technique for creating adaptable applications out of separate components that link together at run time. By implementing a component-based architecture, applications may quickly and easily evolve thanks to the facilities provided by componentization. These include backward and forward compatible upgrades, use of library and purchased components, abstraction and centralization of business logic, dynamic linking, transparent location of components and application customization. We also looked at the alternatives a developer has when adopting a component technology. Whatever technology is adopted, the developer who implements a component-based architecture is designing for evolution.

CASE STUDY: POWERQUERY

PowerQuery is an evolving suite of Internet applications that supports material planning and provisioning for the Columbus GPC, using ODBC, Server-side Scripting, COM, ActiveX and DHTML technologies. Since its inception just two years ago, PowerQuery has gone through four major evolutionary steps. How could this be afforded? How could an application remain in use while so much change was taking place? This was all possible because PowerQuery was explicitly designed to evolve. This section takes a close look at the design principles behind PowerQuery, the same principles discussed in this paper, and demonstrates their effectiveness in creating applications that evolve.

PowerQuery is essentially an application that integrates data from 11 different sources and serves up the results to a web browser. It provides hyperlinked cross referencing, analysis and multi-system views of production, planning and provisioning information. Interactive reports are also a product of the tool, replacing

static paper reports. At fixed weekly, daily and hourly intervals, data is assembled from the various systems. Web pages allow the user to pass parameters on to PowerQuery, which results in a query followed by the display of data. Results sets include active elements that allow users to generate second-level queries by just clicking on a data element. The tool also provides for analysis and standard reports.

PowerQuery started off as an experiment. Wewanted to see if an Internet application was a viable means of allowing users to integrate, manipulate and process planning and provisioning information. If the idea proved viable, then the application was to be rapidly deployed Figure 8 illustrates the architecture of PowerQuery at each stage of its evolution.

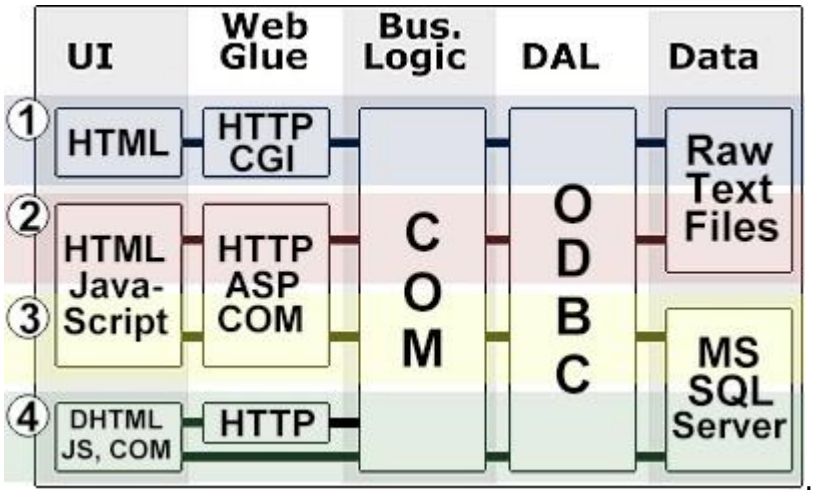

Figure 8

Phase 1 of the application involved using CGI (the Common Gateway Interface), COM components, an ODBC driver for text files and a set of flat ASCII text files. CGI technology was originally used because it was the best alternative at that time. Since new technology was on the horizon, we wanted a solution that would allow us to plug as much of our work into the newly emerging technology as possible. We used COM to create a set of components that would provide the business logic and data services required for the tool. This would allow us to reuse the components with the new technology. It would also allow business logic to be changed without having to change the rest of the application. To keep initial costs and development time low, phase 1 of the application did not involve a DBMS (database management system). Instead, all data resided in a set of raw ASCII text files, many of which were already available. Since we expected to migrate to a DBMS if the application proved viable, we needed to abstract the way data was stored from the data services provided by the application. Rather than accessing the files directly, we accessed them via ODBC, providing the layer of abstraction that we needed. This would allow us to use a DBMS later without having to change our components. The result was a working system that was readily adopted by users. It also evolved into phase 2 very smoothly.

With the success of the phase 1, the decision was made to continue with development. The next steps involved extending the tool's capabilities and use of a DBMS to improve performance. At this time, there was a change in technology known as server side scripting. Server side scripting would significantly decrease development and maintenance cycles. We used Microsoft's Active Server Pages (ASP) to implement the server side scripting in phase 2 which replaced the CGI and added new features. Since our business logic was embedded in its own set of components, the migration to phase 2 only involved the creation of ASP pages that provided the "glue" to connect the existing components with the user interface. The text files were being replaced with a DBMS,

SQL Server, but the work was still in progress after the ASP migration was complete. Since the data services were abstracted using ODBC, we didn't have to wait for the DBMS effort to finish in order to release phase 2. The integration of the DBMS was put off until phase 3 while phase 2 was well received by the users.

Finally, the DBMS was finished. It was loading itself with data from eleven information systems at fixed intervals. Initial testing showed tremendous benefits in both performance and capabilities. Once the DBMS was complete, integrating it with phase 3 was literally a five-minute effort. All we had to do was change the ODBC connection string to reference the DBMS rather than the text files. No other work was required and nothing had to be compiled. Phase 3 was released, being transparent to the users except for the increase in query speed.

At the time of this writing, phase 3 was being replaced by phase 4, a new generation of the tool that provides a dramatically improved user experience. The improvements can be attributed to DHTML and client side components. For these applications, ActiveX is being used as the client component technology. Benefits of phase 4 include the caching of record sets on the client, improved data access and client side data manipulation. Some of the ActiveX components were designed with extensible methods so they won't break old clients when new features are added. The improved solution uses the original server components, so the only work involved in this phase is in the presentation layer and the glue that is used to bind it to the existing business logic. Since most of the original infrastructure remains the same, that is we are only adding to it, it allows phase 3 and phase 4 to coexist. This permits phase 4 to be incrementally deployed with temporary support for older browsers is offered by phase 3, providing a very comfortable and easy evolution. All new work is implemented based on the phase 4 model. To the user, this phase is considered the best overall experience yet.

In this section, we explored a case study on PowerQuery, an Internet application that successfully evolved through four phases of development. The benefits of data abstraction can be seen in use through the tool's implementation of ODBC data access. COM, a component technology leveraged by PowerQuery, which allowed the tool to make major evolutionary leaps without recoding business logic, demonstrates the power of a component-based architecture. We also noted that current development efforts harness ActiveX controls with extensible methods in anticipation of new capabilities—there is no "end state" for this tool. By following the principles of extensibility, abstraction and componentization at design time, PowerQuery has demonstrated that it is entirely possible to deploy solutions that are designed to evolve.

CONCLUSION

Today, business and technology are two very active dynamics that demand every developer's attention. Everything is changing so rapidly. Since developers work in this same fast-paced environment, often building tools to facilitate it, they will either design for evolution or deign for extinction. By employing three fundamental design principles, extensibility, abstraction and componentization, an application may be designed for compatibility with what is yet to come. The end result offers a competitive advantage in the form of an adaptable tool for users and a flexible foundation for developers, moving everyone ahead on the evolution curve.