Progress | OpenEdge

PROGRESS PERSPECTIVE

> IMPLEMENTING THE PROGRESS® OPENEDGE® REFERENCE ARCHITECTURE: OVERVIEW OF DISTRIBUTED ARCHITECTURES

Edition 1.0

Don Sorcinelli

Applied Technology Group

www.progress.com

PROGRESS software

BUSINESS MAKING PROGRESS.

TABLE OF CONTENTS

DISCLAIMER

Certain portions of this document contain information about Progress Software Corporation's plans for future product development and overall business strategies. Such information is proprietary and confidential to Progress Software Corporation and may be used by you solely in accordance with the terms and conditions specified in your PSDN Subscription End User License Agreement. Progress Software Corporation reserves the right, in its sole discretion, to modify or abandon without notice any of the plans described herein pertaining to future development and/or business development strategies.

INTRODUCTION

The Progress® OpenEdge® Reference Architecture (OERA) is intended to provide high-level design guidance focused on building modern competitive applications. Many of the design principals surrounding the OpenEdge Reference Architecture are derived from another architectural concept: service-oriented architecture, or *SOA*. To better understand both the OpenEdge Reference Architecture and SOA, it is important to understand an even higher-level architectural design from which both are derived. Most modern conceptual architectures can find their roots in the design principles behind *distributed architectures*.

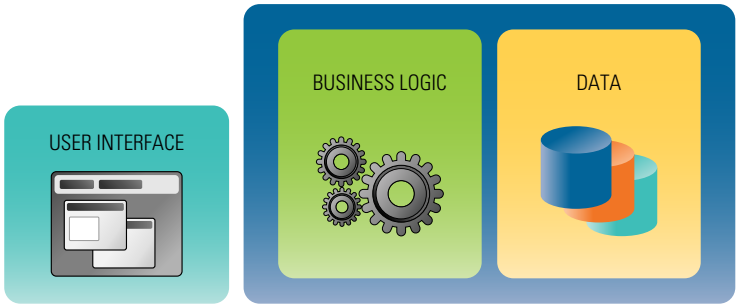
The purpose of this white paper is to provide information and guidance relating to the general concepts and principles of distributed architectures. Information regarding the benefits and challenges will be provided, along with general guidelines for preparing an application to be moved to a distributed architecture. This white paper is intended for those individuals or organizations that are considering developing or transforming an application to a modern competitive architecture but have never been exposed to the design concepts relating to a distributed architecture.

1. AN INTRODUCTION TO DISTRIBUTED ARCHITECTURES

1.1. A BRIEF HISTORY OF APPLICATION ARCHITECTURES

The concept of a distributed architecture is actually the third iteration in the modern history of application architectures. Each general application architecture came into existence in order to both take advantage of new technologies and to address business and technical challenges.

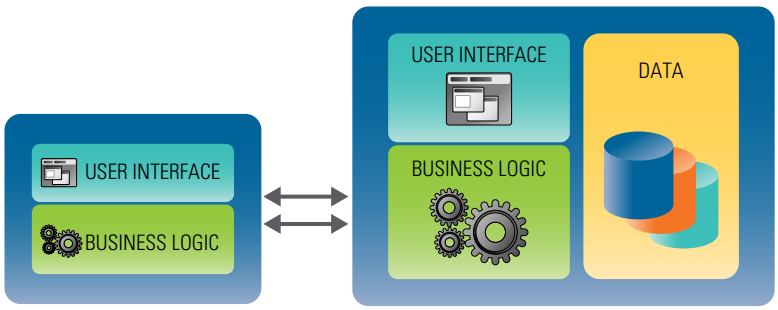
The first modern application architecture was the **host-based** architecture. In this model, all application logic and resources (including databases, external files, etc.) resided on a single physical computer.



Host-based application architectures were highly self-reliant, with little to no need for sophisticated networking technologies. The earliest multi-user applications were based upon host-based architectures. Early personal computing applications were also based upon this architecture, allowing for single-user applications to reside on a single computer without the need for external communication.

While host-based application architectures were initially sufficient for most needs, several issues soon became apparent. First, the reliance on a single computer hosting all resources and user access was not the most scalable means of deploying an application. Large multi-user deployments often taxed the resources of their host computers, and extending the power of these computers was often either not possible or very costly.

Advances in the personal computer space (faster processors, improved graphics and increased memory and storage capacity) and networking technologies (faster and more reliable hardware and networking protocols) became drivers for the second iteration of application architectures. The **client/server architecture** attempted to solve some of the issues of host-based architectures by offloading the processing requirements of an application to any number of personal computers, or *clients.* The host-based computer became the central repository of data (typically in the form of a database) and became known as the *server.*



The distribution of the processing of application logic across numerous client computers often provided a more viable alternative to scaling an application to a larger user base. In addition, client operating systems also provided a number of modern features, including graphical user interfaces and application interoperability, which were restricted or unavailable to host-based operating systems.

While client/server computing provided new benefits to application architecture, it too came with some limitations. Both host-based and client/server applications were often monolithic in their designs. They included an intermingling of user interface and computational and business logic. As a result, these types of applications became difficult to maintain. In addition, the introduction of new user interface environments beyond the character and graphical environment, including browser/web interfaces and the personal digital assistant ("PDA"), often required an entire application redesign and rewrite in order to meet user demands.

Another issue for client/server architectures was scalability in terms of network traffic. With all application logic contained on one computer and a data source contained on another, the movement of large amounts of data across a network could often result in an infrastructure bottleneck. With no complex processing or filtering of data being performed before data was transported to a client machine, what traveled across the network was often times never used within the client application. The shortcomings of previous application architectures eventually led to the introduction of what is now commonly referred to as a "distributed application architecture."

1.2. COMMON ASPECTS OF A DISTRIBUTED ARCHITECTURE

A distributed application architecture, in its purest form, partitions an overall application into logical units for distribution across multiple physical

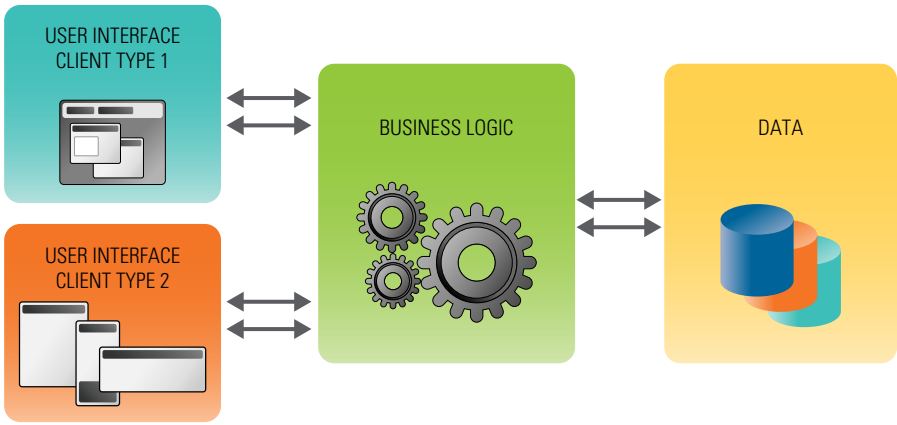
computers. Most commonly, this is accomplished by categorizing application logic into two major groups:

User Interface Logic. This application logic provides the primary interaction point for the user of an application and delivers information using any of a number of appropriate user interface technologies (graphical, web/browser, PDA, etc).

Business Logic. This application logic provides for the processing of information for the application. This includes application logic for business rules and complex analytical processing.

For many, the separation of user interface logic from business logic was an application development best practice for client/server computing. The reason for the separation in a client/server architecture was different, however. Separation of user interface and business logic in a client/server architecture is done almost exclusively in order to maximize code reuse. All application logic runs on a single machine, but the amount of overall code developed and deployed is reduced.

In addition to the benefits of code reuse, the logical partitioning of the user interface and application logic in a distributed architecture allows for the physical separation of the application across multiple computers. This capability has often led a distributed application architecture to be referred to as an *n-tier architecture*.



It is important to note that while a client/server application architecture has a form of physical separation, it is not a distributed

application architecture. The entire application is located on a single computer, with the physical separation between the application and its associated data source(s).

There are several technology requirements that must exist in order for a distributed application architecture to exist. These include:

- **A networking protocol or protocols for communication.** A method of communication must exist between computers hosting application logic in order to facilitate distribution. Today, the most common network protocols are TCP/IP and HTTP (which is actually an additional layer on top of the TCP/IP protocol).
- **A standardized mechanism of communication between the distributed portions of an application.** While a network protocol provides a transport mechanism, there still needs to be a common technology between the distributed application that can process data specific to the programming technology/platform used. The analogy can be made to a phone system and individuals placing calls. While the phone system provides a voice transport between separated individuals, effective communication cannot occur between the two individuals unless they have a common basis for communication (in this case, a spoken language).
- **An application development platform that supports distributed application architecture.** At the minimum, the programming language used for the implementation of a distributed application architecture must support the ability to develop parts of the overall application to communicate across computers and networks to other parts of the application. Additional platform technologies can provide additional benefits supporting the design and implementation of a distributed application architecture.

1.3. DISTRIBUTED ARCHITECTURES AND APPLICATION SERVERS

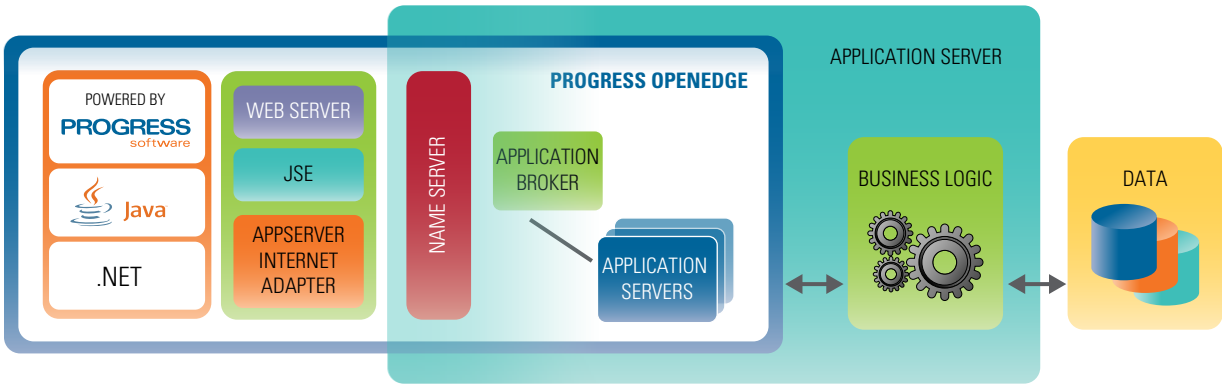
The earliest implementations of distributed application architectures used basic techniques for the communication between application logic across computers. Typically, such implementations included direct TCP/IP socket communication or operating system-supported remote procedure calls (RPCs).

While this solution was technically feasible, it was often difficult to maintain and not easily scalable for a large number of users. In order to effectively support distributed application architectures, a key technology emerged–*the application server*.

The role of an application server in a distributed application architecture is to provide an effective and manageable means for facilitating communication between application logic between computers. An application server usually provides the following services:

- **One or more management processes ("brokers") that facilitate requests to access remote application business logic.** Broker processes serve as a gateway to application server resources, providing the ability to pass requests for application business logic execution to available resources.
- **Processes ("application servers") that execute remote business logic and accept and return data from remote clients.** Remote processes, managed by the application server broker, take requests from remote clients to execute application logic. In addition to application code execution, data is both received and returned.
- **A management interface to configure and control the operation of brokers and application server processes.**

The OpenEdge platform includes the Progress® OpenEdge® Application Server. The OpenEdge Application Server provides all of the functionality mentioned above, as well as providing additional features including load balancing (the ability to dynamically distribute the execution of requests across multiple systems) and multiple connection and operating states (providing flexibility in the configuration and balancing of scalability and state management).



2. BENEFITS OF A DISTRIBUTED ARCHITECTURE

A distributed application architecture provides several distinct benefits for an application developer. Most of these benefits directly relate to solving the problems associated with host-based and client/server application architectures.

2.1. DISTRIBUTING APPLICATION PROCESSING ACROSS MULTIPLE COMPUTERS

Host-based application architectures proved over time to be taxing on a single computer with a large number of users. While client/server architectures relieved the burden off of a single computer, they often simply transferred the same issue to multiple client computers. Large applications with a great deal of complex processing are often just as taxing to all of the individual computers executing the application.

In a distributed application architecture, the application can be partitioned in any of a number of configurations that can more efficiently share the processing workload. This distribution of processing can result in improved application performance.

2.2. PROMOTING CODE REUSE AND STABILITY

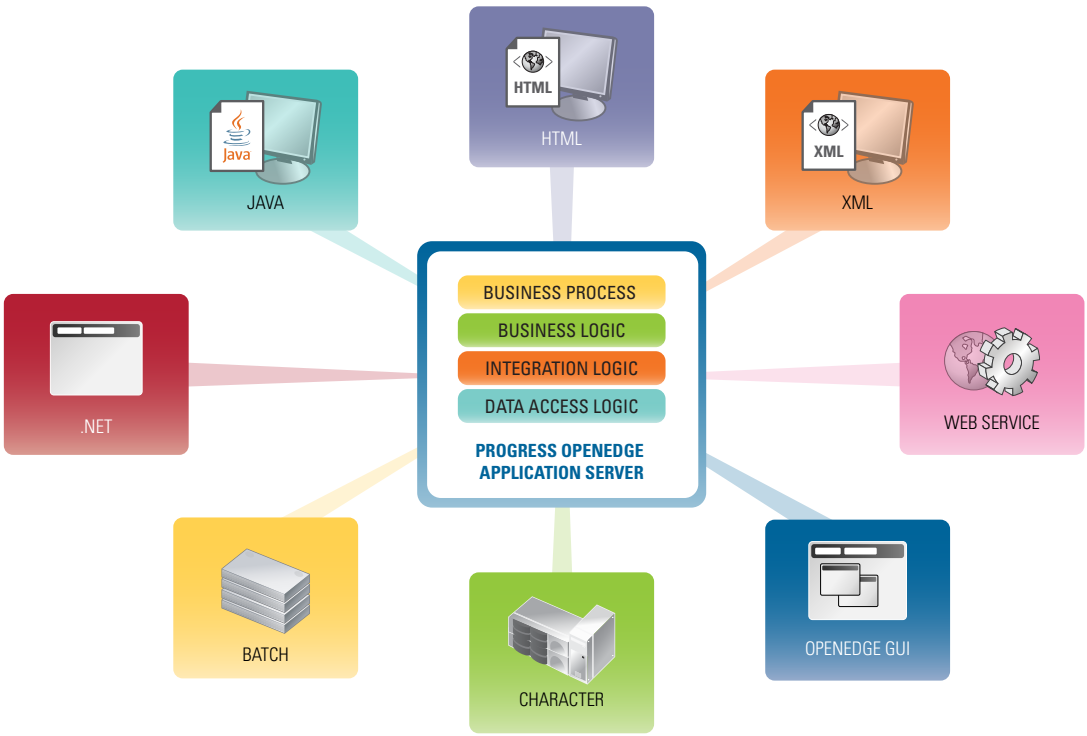
Application business logic typically represents the complex processes associated with the overall application. In addition, this part of the application also represents the foundation of the unique functionality and domain expertise that distinguishes one application from another. Typically, the business logic of an application is stable and not subject to frequent change. As an example, the business rule to compute the sales tax for a product does not change significantly over time. (The data required for the computation may change, but the actual formula does not.) Once developed and tested, business logic typically requires little maintenance or change.

Common business logic is often used throughout an application. The previous example of a tax calculation is no exception. A calculation of this sort can often be used in numerous points within an overall application architecture. While this logic could be repeatedly placed at all points where it is used, this method of code reuse is inefficient and creates maintenance issues. By separating the business logic and placing it in a single location (a separate procedure, for example), the logic can be called when needed to maximize reuse.

2.3. SUPPORT MULTIPLE USER INTERFACE TECHNOLOGIES

As mentioned in a previous section of this paper, the last 15 years have seen the emergence of a number of user interface technologies, including graphical, web/browser-based and PDA. There are still more emerging user interface technologies on the horizon, including ink-enabled and speech. In addition to adapting to new technology, it is often necessary for an application to support these same technologies simultaneously.

While it is possible to recreate all application logic for each of the user interfaces required, a more efficient approach is to centralize all common business logic in a distributed architecture and have each of the user interface types access this common logic through an application server.



When combined with the stability of application business logic, a distributed architecture can allow for a focus on the development of new user interfaces that access the common logic.

2.4. SUPPORT INTEGRATION FUNCTIONALITIES

Thus far, all of the discussions surrounding an interface for an application have focused on a user interface. Modern competitive applications are, for both business and technical reasons, requiring the ability to provide application integration functionality. While it has been noted that a well-designed distributed application architecture is "user interface-agnostic," it can also be noted that this very same type of architecture can be "*all* interface-agnostic." Application integration is, in essence, a form of non-user interface.

By properly separating business logic and placing that logic into a distributed architecture, the business logic can be exposed to other applications for integration purposes in any of a number of ways, including the use of cross-platform technologies like web services, which are supported in the OpenEdge platform.

3. DESIGN CHALLENGES FOR DISTRIBUTED ARCHITECTURES

While a distributed application architecture provides a number of advantages over host-based and client/server architectures, it is not without its own unique design challenges. In planning for the implementation of a distributed architecture, these challenges will need to be addressed.

3.1. INDIRECT DATABASE ACCESS

Host-based and client/server applications typically interact with database data directly within the user interface. Consider the following code that can be found within a user interface trigger–

```
ON CHOOSE OF bNEXT IN FRAME {&FRAME-NAME}
DO:
        FIND NEXT Customer NO-LOCK NO-ERROR.
        DISPLAY
                Customer.CustNum
                Customer.Name
                WITH FRAME {&FRAME-NAME}.
END.
```

This user interface logic assumes that the client portion of the application is directly connected to the database. In a distributed application architecture, it is usually the business logic portion of the application that manages database connectivity and access. As a result, it will be necessary to re-think and redesign the overall application logic to compensate for indirect database access.

The Progress Advanced Business Language (ABL) does provide for the passing of database data using several methods, including:

- **Variables.** Standard variables can be used when simple data is required to be passed to and from business logic.
- **Temp-tables.** Temp-tables are a temporary representation of table-based data. Interactions with a temp-table are similar to that of a regular database table.
- **Progress® ProDataSet™.** A ProDataSet is another temporary representation of database data. It can be used to hold both data and relationships for multiple tables.

Passing data using any of the three methods referenced above is the most effective way to manage indirect database access within the user interface.

3.2. TRANSACTION MANAGEMENT

When directly accessing a database from a user interface using the Progress ABL, it is very easy to control and manage transaction scope for database changes. As mentioned previously, a distributed application architecture typically does not have direct database access from the user interface. As a result, transactions cannot be directly managed from the client portion of the application. Please note that this does not mean that transactions as a whole cannot be managed. All of the transaction management services built into the RDBMS and programming language are still available, but only to the portion of the application logic that is actually accessing them directly (in this case, business logic residing on the server side of the application architecture). It is the client portion of a distributed architecture that does not have this direct management capability.

In typical distributed application architectures, transaction management is resolved using the following high-level sequence:

- Data is passed to business logic for processing.
- The business logic manages the transaction, performing any processing and validation.
- If errors occur as a result of the business logic, the business logic returns an error and passes appropriate information back to the client.
- The client responds appropriately depending on application rules (error message, data display, etc.).

In preparing for a distributed architecture, planning and design for transaction management will need to be carefully considered.

3.3. CONTEXT MANAGEMENT

In a distributed application architecture, parts of the application are running on different computers and, therefore, running in separate memory spaces. As a result, application data is not automatically shared. In designing a distributed application, considerations are necessary for frequently passing important information between user interface and business logic.

One example of context management is user information. A user ID is commonly used for authentication and authorization regarding application functions. In a host-based or client/server architecture, this data is typically stored once, then accessed as needed using a shared resource (a shared variable, for example). In a distributed architecture, however, business logic running on a separate computer will not have direct access to the memory resources on the client computer. As a result, this information will likely need to be passed multiple times during an application session.

4. CONSIDERATIONS FOR MOVING TO A DISTRIBUTED ARCHITECTURE

In order to properly transform an application from a host-based or client/server architecture to a distributed architecture, there are a number of actions that will need to be performed. It is important to note that the

process of an application transformation is typically quite lengthy and includes application analysis and design in addition to code migration (or *harvesting*). In Section 5 of this paper, there is information relating to learning more about the refinement of analysis and design for building modern business applications. The information that follows in this section should be considered as a baseline for starting to transform an application to the most basic of distributed architectures.

4.1. IDENTIFYING EXISTING BUSINESS LOGIC

In analyzing an existing application, a key goal is to harvest business logic: that is, to try to identify reusable business logic that is suitable for reuse. Some general considerations for identifying business logic include:

- **The application logic does not have a direct dependency on user interaction.** Much of an application's code does not directly interact with a user interface. Instead, it either relies upon or works in conjunction with user interaction. Typical Progress ABL statements in this category include:
  - Data retrieval (FOR EACH, Query-related)
  - Data manipulation (CREATE, DELETE, ASSIGN between objects)
  - Rules processing
- **The application logic is used at more than one point in the application.** Typically, business logic is used in multiple points in the overall application architecture. The more frequently used a piece of application code is, the more desirable this code will be as a candidate for harvesting.

In addition to analyzing code in Progress ABL procedures, it is also important not to overlook other potential sources of business logic. These sources include ABL include files and database triggers.

4.2. SEPARATING BUSINESS LOGIC

At an absolute minimum, business logic will need to be separated from user interface logic and placed into new Progress ABL procedures that can be distributed across computers. There are two basic approaches to doing this:

1. **Creation of individual procedures based upon distinct business logic functions.** In this scenario, each single piece of business logic identified would be placed into a new Progress ABL procedure. The business logic would then be accessed from the client portion of the application using a RUN statement.

2. **Creation of a single procedure that includes multiple business logic functions.** In this scenario, a single Progress ABL procedure would consist of multiple internal procedures and/or user-defined functions. Each of the internal procedures would contain a specific business logic function. Typically, each ABL procedure would contain related business functions grouped in some logical fashion. An example of this would be a ABL procedure that contained a number of business logic functions relating to a shipping activity. Internal procedures might include:
   - Calculating shipping price
   - Printing shipping labels
   - Printing a packing slip, etc.

For this scenario, the client portion of the application will need to perform two steps in order to access the appropriate business logic. First, the ABL procedure will need to be run persistently. Then, internal procedures can be invoked as needed within the persistent procedure.

4.3. ENCAPSULATING BUSINESS LOGIC

In order for a distributed application architecture to be effectively implemented, business logic must be able to communicate with a client application. In order to do this, steps must be taken to make Progress ABL procedures both communicative and self-reliant. In essence, ABL business logic procedures must be encapsulated in order to work properly in a distributed application architecture.

There are several things to consider when creating and/or migrating to encapsulated business logic:

- **There can be no direct programmatic reliance on shared resources from the client portion of the application.** It is important to remember that the client portion of the application and the business logic run on separate computers in separate memory spaces. As a result, Progress ABL SHARED objects (shared variables, frames, buffers, etc) cannot be used. Instead, this information will need to be passed from client to business logic using parameters. From a best-practices perspective, it is recommended that use of shared objects be kept to an absolute minimum (if at all) in modern applications. The reason for this is fairly straight-forward. Shared objects require the initial creation of the object in one ABL procedure (DEFINE NEW SHARED…) before other references can be made (DEFINE SHARED…). In strictly procedural programming, the order of processing is tightly controlled, and the order of object instantiation is very predictable. In event-driven programming models where logic and procedures are less controlled and predictable, it is possible that a reference to a shared object would be made before it was first instantiated with the new shared ABL syntax. This type of dependency goes against the principles of encapsulation and results in application execution issues.

- **Direct database access will reside in encapsulated business logic, and database information will be passed to and from the client.** Both business logic and client logic will have to compensate for the lack of direct database access from the user interface. As mentioned earlier in this paper, this can be accomplished by using Progress ABL variables, temp-tables and ProDataSets. These objects can be populated from the business logic, and then passed to the client portion of the application using parameters. This data would be interacted with by the user, and the results passed using the Progress ABL RUN statement and parameters.

- **Some data will need to be frequently passed as part of context management.** Previously in this paper, the issue of context management was discussed. Because of the disconnected state of a distributed architecture and business

logic encapsulation, steps will need to be taken in order to effectively pass and manage key context information. In the previous reference to context management, a user ID was cited as likely to be passed frequently for authorization and authentication. This information is typically quite small and results in little network or application overhead. In certain situations, however, a larger amount of contextual information may need to be managed. This is most common in cases where a web browser is the foundation of a user interface and no true mechanism exists for holding large amounts of contextual information.

There are several approaches to dealing with context management. One possible solution, for example, is to pass a single unique identifier between the client and business logic. This unique information (like a user ID) serves as a key for storing additional context information in a database for the life of a user session. As more context data is created, it is stored in the database using the user ID as a key field. When needed, application business logic retrieves the information. When the user's session completes, the context data for the session is deleted.

There are other variations on this method that can be employed, but the same principle still exists–context management will likely need to be implemented for some portion of an overall distributed application architecture.

4.4. REFINING BUSINESS LOGIC

It is quite feasible to simply remove business logic from an existing Progress ABL procedure, encapsulate it in a new ABL procedure and modify the client portion of an application to access it. In this case, distributed functionality has been created. It may not, however, be the best solution towards achieving an effective distributed architecture.

In the introductory section of this paper, it was noted that the design concepts behind service-oriented architecture ("SOA") and the OpenEdge

Reference Architecture were derived from the basic principles of distributed architectures. These newer architectural design concepts are not just descendents of distributed architecture; they are also enhancements and refinements of original design concepts based upon learned experiences and changes to both technology and the software industry.

Both SOA and the OpenEdge Reference Architecture encourage developers to think of their application logic as components in a larger application architecture. The goals behind these components are to address and anticipate business and technical challenges associated with modern application applications. Furthermore, the OpenEdge Reference Architecture refines the definition and purpose of various types of Progress ABL components that can exist within an application architecture. Rather than grouping blocks of application logic together simply for the ability to distribute, the OpenEdge Reference Architecture helps to organize logic by both function and the role of the logic in the larger architecture.

By further defining and refining the structure of business logic in a distributed architecture, greater flexibility and code reuse can occur. The refinement can also open additional application opportunities. These opportunities include facilitating the use of web services and enabling robust integration capabilities.

Before a move towards a distributed application architecture, it is strongly recommended that effort be put forth to learn more about the ways in which a distributed application architecture can go beyond simple distribution of application logic.

PROGRESS SOFTWARE

Progress Software Corporation (NASDAQ: PRGS) is a global software company that enables enterprises to be operationally responsive to changing conditions and customer interactions as they occur. Our goal is to enable our customers to capitalize on new opportunities, drive greater efficiencies, and reduce risk. Progress offers a comprehensive portfolio of best-in-class infrastructure software spanning event-driven visibility and real-time response, open integration, data access and integration, and application development and management—all supporting on-premises and SaaS/cloud deployments. Progress maximizes the benefits of operational responsiveness while minimizing IT complexity and total cost of ownership.

WORLDWIDE HEADQUARTERS

Progress Software Corporation, 14 Oak Park, Bedford, MA 01730 USA
Tel: +1 781 280-4000 Fax: +1 781 280-4095 On the Web at: www.progress.com

Find us on facebook.com/progresssw twitter.com/progresssw youtube.com/progresssw

For regional international office locations and contact information, please refer to the Web page below:
www.progress.com/worldwide

Rev. 12/11 | 111202-0103