feature stories

The Shape of Things to Come

Using Patterns to Develop Distributed Applications

by D. William Reynolds, Jr.
Founder and President, Austin Software Foundry, Austin, Texas

Success in any field depends on making the most of existing knowledge, experience, and results–shaping this information into reusable "components" that can be used to create new, more complex structures. Software engineers, however, work in a maturing field. While mechanical, electrical, or chemical engineers follow the laws of chemistry and physics to create bolts and threads, solvents, and integrated circuits, software engineers don't have as many immutable laws and principles to follow. On what guidelines can they rely for creating complex, distributed systems? A significant part of the answer lies in the recognition and use of existing software patterns, much like other engineers rely on the existing laws and principles that govern their fields, to create and combine the building blocks of a distributed application.

Though the idea may sound new, a large body of knowledge related to the development of distributed software patterns does exist. In the first installment of this two-part article (available at http://www.sybase.com/inc/powerline/powerline_q298/shape.html), I will introduce you to the use of the industry-standard patterns that I use and to how they will help you produce applications with PowerStudio™–Sybase's suite of best-in-class development tools–that leverage the principles of software engineering while meeting the real-world demands of your customers. This first article presents four of ten patterns that you can then combine into the foundation framework of a distributed application. The following six patterns and their use with PowerStudio will follow in the July issue of *Powerline*–which will be completely online. Ultimately, you will have a framework for class combinations that you can refer to while you develop that will guarantee a distributed application.

An Easy Pattern of Success

The General Idea

Most distributed application development efforts face a common set of problems that have been solved before. These problems might include user interface navigation, inventory management, communication, command processing, adaptation, or transaction management. Principles like structured programming, modularity, coupling, cohesion, correctness, and object orientation reflect the concepts that address such issues. Software patterns provide developers with concrete examples of how to combine and apply these software principles to quickly and efficiently solve these and more complex development problems.

Like recipes, patterns show how to combine classes and objects into components, frameworks, and applications that are based on these proven principles. Typically, a pattern explanation presents a solution to the development problem in the form of a programming-language-neutral model–most often an object model–followed by an example of pattern application in one or more programming languages. A development team with access to and knowledge of these patterns is much more likely to deliver a quality solution on time and within budget than a team facing these problems without the benefit of prior solutions.

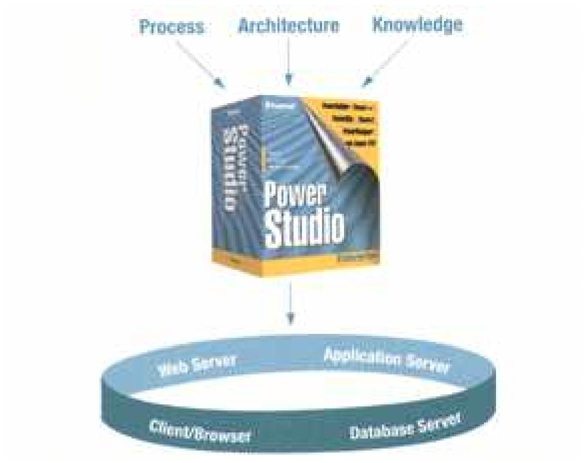

Figure 1: The application of established development processes, architectures, and knowledge in the shape of patterns to distributed development allows developers to make the most of sophisticated development tools like PowerStudio. The result–efficient creation of complex, distributed applications.

A Closer Look

But why do we need to discuss using patterns at all? Isn't addressing and solving these development problems just part of the development process? The truth is that while objects and components are rapidly becoming the foundation to most approaches to creating distributed applications, object and component use presents significant challenges to many development organizations. Using objects to build applications results in decentralizing the flow of control and requires managing complex structural relationships involving inheritance hierarchies and the composition of components from many finer-grained elements.

Additionally, while in object-oriented development tools (PowerBuilder®, PowerJ++™, and Power++™ for example, component pieces of PowerStudio) the finest-grained level of development yields classes and objects, it's difficult to reuse previous work or add new team members who are unfamiliar with object-oriented development because classes and objects are so small.

Many teams now focus on reusing components, frameworks, and even whole application templates to eliminate some of the complexity. The creation of these "coarser grained" components and frameworks is based upon patterns.

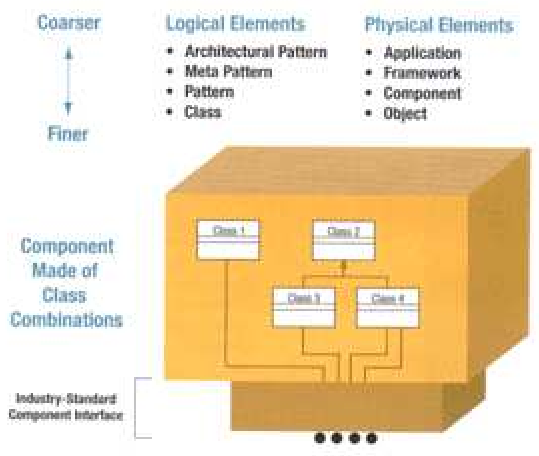

Figure 2: The combination of finer-grained elements such as classes and objects into coarser-grained elements, such as components, heightens code reusability. Component creation, for example, facilitates broad use of your code by allowing the attachment of an industry-standard interface.

Fundamental Patterns for Developing a Distributed Business System

Software patterns fall into three basic categories of functionality: application patterns, domain patterns, and technology patterns. Within those categories, individual patterns address specific pieces of functionality.

Application Patterns

Application patterns, also called user-interface patterns, present combinations of windows, controls, menus, and other graphic elements that solve common problems like navigation, screen layout, and data presentation–problems associated with user-interface implementation and with the functionality needed within a specific application to support the client-side presentation of a dynamic, graphical user interface. In short, they help you design the implementation of the client side of a distributed application. Familiar examples of application, or user-interface, patterns include Multiple Document Interface and Master Detail patterns.

Multiple Document Interface (MDI) Pattern

One of the first design issues faced by most software development projects is how to present the system to the user in a way that is intuitive and integrates with the underlying business process or work flow. A common problem faced by many business applications is that multiple windows must be open simultaneously, each containing different information or different views of the same information. For example, a sales person may need to open a window to enter a new order and then access information on several different products while completing the order.

The MDI pattern reflects a common solution to this problem. The pattern consists of a frame window and menu that contain and manage a series of sheet windows. The frame menu provides the shell for multiple sheets in the application and acts like a main window. The sheets each provide the user with a view of the underlying information and data-processing functionality. PowerBuilder's Window Painter has long provided an implementation of this pattern, but the pattern can be employed just as easily in Java or C++ development–including with Sybase's PowerJ and Power++ development tools.

Master Detail Pattern
Another common user interface problem faced by business applications is that one piece of information displayed in one user interface object or control–the master object–has a one-to-many relationship with information displayed in another object or control–the detail object–within the user interface. For example, a window control shows a list of directories. When a directory is selected, another control shows all files in the directory. The problem is that if the contents change for the master object–the window control–then the contents need to change for the detail object, the control that shows directory contents.

The Master Detail pattern solves this problem by expressing the relationship between a master user interface object and a detail user interface object. This pattern consists of a master data control object and a detail data control object. The master object displays the master data values, keeps track of user selections among these master data items, and requests the detail information for the master items selected by the user. The detail object accepts a request from the master object and displays detail information based on a key provided by the master object.

Domain Patterns
For solving problems in the business domain and guiding implementation of the business logic on an application server, domain patterns provide the answer. These patterns present combinations of nonvisual classes that represent business objects–like customer, product, transaction, manager, or employee–that solve common problems, such as how to track the movement of products into and out of inventory, how to represent the multiple roles that people play in an organization, or how to design the behavior required for a customer to purchase something from a supplier. In a distributed application, these patterns are drawn from data-modeling experiences in different domains, such as accounting and inventory-tracking systems. The core concepts behind these business systems, such as having caches of money and goods and how they are recorded and moved to various reserves, can be abstracted to create domain patterns for the accounting and inventory domains.

Item-Item Description Pattern
The Item-Item Description pattern is a basic inventory and accounting pattern that helps eliminate redundant information associated with large inventories of identical products with complex descriptions. Such inventories might include a fleet of Boeing 727 airplanes or a warehouse full of identical refrigerators. The item object represents the unique items in the inventory, while the item description object represents a description common to all the identical items.

Technology Patterns
Most pattern-identification effort to date has been in the area of technology patterns, which help solve problems specifically related to software development and system infrastructure design. Specifically, technology patterns present combinations of nonvisual classes that represent technology objects–like collection, iterator, adapter, facade, or controller–that solve problems such as how to sort a collection of objects, how to communicate with an external or legacy computer system, or how to send messages across a distributed architecture between the user interface and the business objects on an application server. The Model-View-Controller pattern offers a critical architecture for distributed applications–the architecture that separates the user interface from the business logic.

Model-View-Controller Pattern
Because the user interface is prone to changes, designers often want to design a partitioned application that gives them the flexibility to physically separate the user interface from the application logic. The user interface can then be modified, extended, or distributed independently of the application logic.

The Model-View-Controller pattern addresses this challenge by allowing developers to divide an interactive application into three types of components. You will find this pattern used successfully in PowerBuilder, C++, and Java development. The Model object contains the business logic–the functional core of the application. Model objects encapsulate the business data and perform application-specific processing. The view object is a collection of windows that display information to the user, accept input from the user, and translate that input into requests sent to the controller. The controller acts as a translator or intermediary between the view and the model, creating and initializing the view and model objects and sending requests back and forth between them.

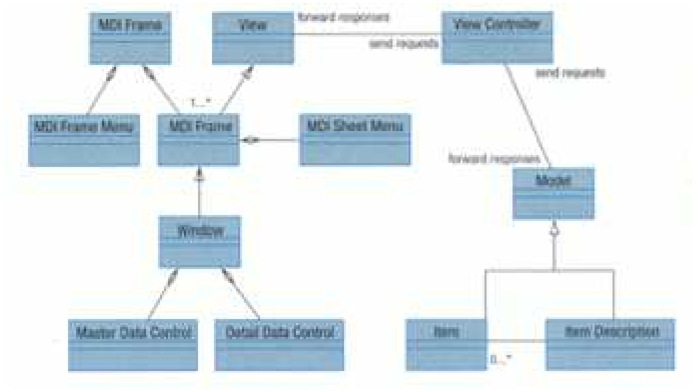

Figure 3: You can combine classes according to the four patterns presented, and then combine those architectures to create the most fundamental framework of a distributed application.

Looking Ahead
The four patterns I've presented can be combined into the foundation of a distributed application framework and can be used with a tool set like PowerStudio to create distributed applications.

In the second part of this article, available in *Powerline's* July issue, I'll introduce several more technology and domain patterns to flesh out our application framework. These will include domain patterns for implementing use cases and managing multiple business contexts with a single application, and technology patterns for handling messaging, event broadcasting and notification, separation of interface and implementation classes, and integration with existing or external systems or subsystems. Finally, I'll use the framework we have created to demonstrate how a complete tool suite like PowerStudio can be used to implement distributed applications from these common architectural principles and patterns.

AUTHOR:
D. William Reynolds, Jr. is the founder and President of Austin Software Foundry in Austin, Texas. Austin Software Foundry has been a Powersoft/Sybase partner since 1992 and authored the Object-Oriented Analysis and Design with PowerBuilder,

Object-Oriented PowerBuilder Development, and Application Partitioning with PowerBuilder courses. Bill has a B.S. in Finance from The University of California at Berkeley and an MBA in Information Systems Management from The University of Texas at Austin. He will present many of the concepts and techniques from this article at Powersoft Conference '98 in Los Angeles in a course called Architecture for Object-Oriented Development.