Chapter 1

Introduction to Enterprise Software

If you have heard of terms such as Business-to-Business (B2B) and Business-to-Consumer (B2C), you are already familiar with enterprise software at some level. B2B and B2C are just some of the more popular manifestations of enterprise software.

This introductory chapter offers a more in-depth exploration of enterprise software and the challenges and opportunities that accompany it.

What Is Enterprise Software?

The term *enterprise* refers to an organization of individuals or entities, presumably working together to achieve some common goals. Organizations come in all shapes and sizes, large and small, for-profit and nonprofit, governmental and nongovernmental.

Chances are, however, that when someone uses the term enterprise, they mean a large, for-profit organization, such as Intel, General Motors, Wal-Mart, Bank of America, or eBay.

Enterprises generally have some common needs, such as information sharing and processing, asset management and tracking, resource planning, customer or client management, protection of business knowledge, and so on. The term enterprise software is used to collectively refer to all software involved in supporting these common elements of an enterprise.

Figure 1-1 depicts enterprise and enterprise software graphically.

The figure shows an enterprise software setup that is essentially a collection of diverse systems. Software is organized along the various functions within the organization, for example, sales, human resources, and so on. A firewall is provided to safeguard enterprise data from unauthorized access. Some software systems such as those for sales and inventory management interact; however, most are fairly isolated islands of software.

Enterprise software may consist of a multitude of distinct pieces today, but enterprises have gradually come to realize that there is a strong need for their diverse systems to integrate well and leverage each other wherever appropriate for maximum enterprise benefit. B2B and B2C are good examples of such integration and leveraging.

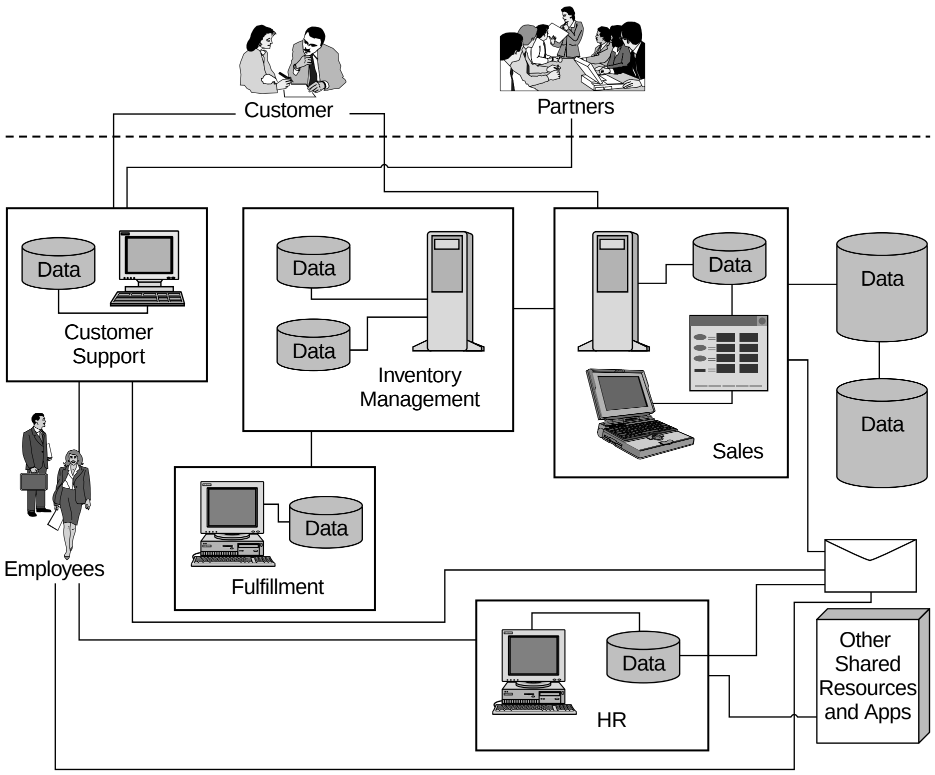

Figure 1-1 Enterprise and enterprise software

Some of the potential ways an enterprise hopes to leverage integrated enterprise software follows:

- By integrating its customer support and in-house product knowledge, an enterprise could provide new and better services to its customers via the Web.
- By linking its marketing machine with the online world, an enterprise could reach a much larger audience online.
- By linking its sales management and inventory, an enterprise may be able to devise specific, lower cost Web sales channels to reach an untapped market segment.
- By providing a front end to one of the services used by its employees, such as the internal office supply ordering system, and tying it into the account-

ing system, the enterprise could lower the overall cost and improve employee efficiency.

- Making the enterprise HR system available online could be used as a way to give employees more control over their health and 401(k) choices and reduce the overall administrative costs to the enterprise.
- By automating one of its human resource intensive operations and making it available on an anytime, anywhere basis, an enterprise could provide better service to its customers while reducing the overall operational costs.

Challenges in Developing Enterprise Software

Successful enterprises tend to grow in size, hire more people, have more customers and more Web site hits, have bigger sales and revenues, add more locations, and so on. In order to support this growth, enterprise software must be scalable in terms of accommodating a larger enterprise and its operations.

Enterprises encounter constraints as they grow. One common constraint is the computer hardware's inability to scale as the enterprise's processing needs increase. Another constraint is the enterprise's ability to put more people in the same physical or even geographical location. Thus, the challenge of distribution comes into the picture. Multiple physical machines solve the processing needs but introduce the challenge of distributed software. New building or geographical locations address the immediate need, but they introduce the challenge of bringing the same level of services to a diversely located enterprise.

Connecting previously separate systems in order to gain enterprise-scale efficiencies can be a major challenge. Legacy systems were typically designed with specific purposes in mind and were not specifically conceived with integration with other systems in mind. For example, human resource management perhaps was treated as a distinct need without much interaction with financial management, and sales management had little, if anything, to do with customer support. This disjointed approach to software development often resulted in excellent point products being purchased to address specific needs, but it commonly resulted in software architectures that were difficult to integrate.

A related challenge is the need to deal with a multivendor environment. Partly out of evolution, and partly out of necessity, enterprise software has often ended up with similar products from multiple vendors used for the same purpose. For instance, although the HR application might be built on an Oracle 8i database, the customer support application might rely on Microsoft SQL Server.

Enterprise software also typically requires some common capabilities, such as security services to safeguard the enterprise knowledge, transaction services to guarantee integrity of data, and so on. Each of these requires specific skills and knowledge. For instance, proper transaction handling requires strategies for recovering from failures, handling multiuser situations, ensuring consistency across transactions, and so on. Similarly, implementing security might demand a grasp of various security protocols and security management approaches.

These are just some of the common challenges that must be addressed when dealing with enterprise software development.

Evolution of Enterprise Software

Not too long ago, mainframes ruled the world, and all software was tied to this central entity. The advantages of such a centralized approach included the simplicity of dealing with a single system for all processing needs, colocation of all resources, and the like. On the disadvantage front, it meant having to deal with physical limitations of scalability, single points of failure, limited accessibility from remote locations, and so on.

Such centralized applications are commonly referred to as *single tier* applications. The Random House dictionary defines a *tier* as "one of a series of rows, rising one behind or above another." In software, a tier is primarily an abstraction and its main purpose is to help us understand the architecture associated with a specific application by breaking down the software into distinct, logical tiers. See Chapter 6 for a more detailed discussion of tiers.

From an application perspective, the single most problematic aspect of a single tier application was the intermingling of presentation, business logic, and the data itself. For instance, assume that a change was required to some aspect of the system. In a single tier application, all aspects were pretty much fused; that is, the presentation side of the software was tied to the business logic, and the business logic portion had intimate knowledge of the data structures. So any changes to one potentially had a ripple effect and meant revalidation of all aspects. Another drawback of such intermingling was the limitations it imposed on the reuse of business logic or data access capabilities.

The client-server approach alleviated some of these major issues by moving the presentation aspects and some of the business logic to a separate tier. However, from an application perspective, the business logic and presentation remained very much intermingled. As well, this *two-tier* approach introduced some new issues of its own, for instance, the challenge of updating application software on a large number of clients with minimal cost and disruption.

The *n-tier* approach attempts to achieve a better balance overall by separating the presentation logic from business logic and the business logic from the underlying data. The term *n-tier* (as opposed to *three-tier*) is representative of the fact that software is not limited to three tiers only, and can be and indeed is, organized into deeper layers to meet specific needs.

It should be noted that each tier in an n-tier does not imply a separate piece of hardware, although that is certainly possible. A tier is, above all, a separation of concerns within the software itself. The different tiers are logically distinct within the software but may physically exist on the same machine or be distributed across multiple machines.

Some examples of the types of advantages and benefits offered by n-tier computing are

- *Faster and potentially lower cost development:* New applications can be developed faster by reusing existing, pretested business and data access components.
- *Impact of changes is isolated:* As long as interfaces remain unchanged, changes on one tier do not affect components on another tier.
- *Changes are more manageable:* For example, it is easier to replace one version of a business component with a new one if it is residing on a business tier (on one or a few dedicated servers) rather than having to replace hundreds or thousands of client applications around town, or around the globe.

Figure 1-2 illustrates enterprise software organized along these single, two, and n-tiers.

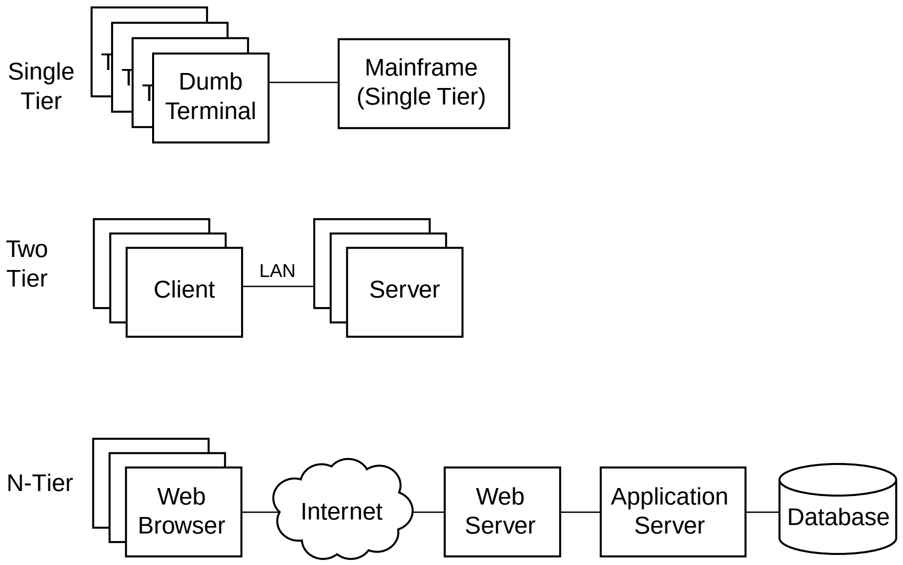

Figure 1-2 Architectural evolution of enterprise software

Enterprise Software and Component-Based Software

When the object-oriented software approach burst onto the software development scene, it was widely expected that adoption of object-oriented software development techniques would lead to reuse, but this hope was only partially realized. One of the reasons for this partial success was the fine granularity of the objects and the underlying difficulty of achieving large-scale reuse at that level due to the more strongly coupled nature of fine-grained objects.

Software components are designed to address this precise issue. Unlike an object, a software component is designed at a much higher level of abstraction and provides a complete function or a service. Software components are more loosely coupled. Using interfaces the components have deliberately exposed, they can be combined together rapidly to build larger applications quickly and are more cost-effective.

Component-based software, of course, requires that components from different sources be compatible. That is, an underlying common understanding, a contract if you will, is required on which the components are to be developed.

Various *component models* have been developed over the years to provide the common understanding. Microsoft's ActiveX, later COM, and Sun Microsystem's applets and JavaBeans are examples of such component models.

Distributed component models have also been developed to address component-based software in the context of distributed enterprise software and associated challenges discussed earlier. Such component models essentially provide an "operating system" for distributed and component-based software development. Examples of these include DCOM, Microsoft DNA (now Microsoft.NET), and Sun Microsystem's Enterprise JavaBeans (EJB), which is part of the Java 2 Platform, Enterprise Edition (J2EE).

Summary

Enterprise software has undergone a gradual evolution in pursuit of providing ever-greater value to the enterprise. Enterprise software faces some distinct challenges. These include, among others, scalability, distribution, security, and the need to work with a diverse set of vendor technology. Various evolutionary architectural approaches have been tried over the years to meet such challenges. An increasingly popular solution revolves around using a distributed component model to develop superior enterprise software. Such distributed component models hold promise, but they are still in their infancy.