REUSE OF SOFTWARE MODULES

Kathryn J. Anderson, Roger P. Beck, and Thomas E. Buonanno

__Kathryn J. Anderson__ is director of the UNIX® System Application Environment Laboratory at AT&T Bell Laboratories in Summit, New Jersey. She is responsible for planning and development of graphics products, programming languages and utilities, database management systems, and transaction processing software. She received a B.S. in mathematics from Arizona State University and an M.S. in computer science from Rutgers University. She joined AT&T in 1973. __Roger P. Beck__ is supervisor of the Systems Software Development Group in the Software Technology Development Department at Bell Laboratories in Columbus, Ohio. He is responsible for development of many of the reusable modules discussed in this paper. He received a B.S. and an M.S. in electrical engineering from
(continued on page 76)

Reusing software that was originally developed for another application increases the productivity of software developers and improves the quality of software applications. A further benefit of reuse is that software standards, once implemented, can be employed many times. As a result, users perceive a resemblance among different applications and learn to operate them more quickly and at lower training cost. This paper describes a group of compatible, integrated, reusable software modules. It discusses recent experience with reusable modules in two projects and the productivity improvements that accompanied them.

Introduction

For some time, software developers have been concerned about the rising cost of building and maintaining software applications. There is a pressing need to increase the productivity of people who build software applications and to improve the quality of applications.

Among the most promising alternatives for increasing productivity and quality and decreasing implementation and maintenance cost is software reuse—the use of software that was originally developed and deployed for another application. By this definition, developing a utility library (for example, data manipulation routines) within an application does not constitute software reuse, even though it increases productivity. On the other hand, adding an existing user interface or error logger is considered software reuse.

In addition to productivity and quality improvements, another important benefit accrues from software reuse: standards can be implemented once and reused many times. For example, if several applications reuse the same software standards for the interface with the end user, then the "look and feel" of the applications will be the same. Users will perceive a family resemblance, and will learn to operate new applications in the family quickly and at lower training cost.

Reuse of large-scale software components, or modules, is particularly productive. Reuse of several integrated modules is even more productive.

Our recent experience with integrated reusable modules in software development projects shows that the productivity improvements are real and substantial. We estimate that staff effort was reduced to about half the effort that would have been required without reuse.

Effective module reuse requires that the relevant software:

- Be useful to a range of applications
- Be adaptable readily to the applications that use it
- Provide adequate performance and reliability
- Meet standard interfaces.

This article describes the types of software components that form integrated reusable modules.

Application Modules

A review of the evolution of applications built on the UNIX® operating system will illustrate the effectiveness of module reuse.

Evolution of Module Reuse. Before the late 1970s, AT&T applications were built either on large mainframes or on assorted minicomputers with homegrown operating systems. In the late 1970s, AT&T developers began to build applications on the UNIX operating system. Since the UNIX operating system previously had been used just for general-purpose time-sharing, it lacked software components for database management, transaction monitoring, and user interfaces. For the most part, the developers of AT&T applications built their own application architectures. Software developed by one project was specialized for the needs of the local organization, so reuse of others' software was rare.

As the development process matured, it became clear that developing production-quality applications requires a well-defined architecture. Within such an architecture, module reuse can be implemented. One of the first of these modules was the AT&T Tuxedo™ transaction processing system,[1] which was used in several projects and augmented with other functions commonly needed.

Architecture and Module Reuse. Architecture, a high-level description of the overall structure of an application, defines the major application components (hardware and software) and their interrelationships. Architectures, therefore, contain application-specific information. For software, the architect must:

- Define a robust *decomposition* of the application—that is, break it down into well-defined functional components. Each component is distinct from every other component, performs a specific application task, and has specific communication interfaces.
- Specify the interfaces and protocols for each of the components in the decomposition.
- Describe the behavior of each component of the decomposition—that is, describe the changes in the application's state or the effects on other components that result from the execution of each component. If critical, the performance of the component may also be characterized.

Using standard approaches to these activities leads to improved opportunities for module reuse and, thus, to higher productivity. The discipline of standard process design (functional decomposition) and standard interface and protocol definition encourages workers on a project to develop generalized or easily generalizable modules whenever possible, so that workers on other projects can find and adopt components for their applications. With common software, developers can transfer more easily between projects and carry ideas with them. With a standard database management system and a common data dictionary, for example, applications can share data easily.

Other Benefits of Module Reuse. Most of the productivity and quality improvements of reusable modules result directly from reuse. However, additional benefits, such as faster development, result from the development process itself. The following sections give an overview of how developers can build an initial version of the application early with the help of integrated reusable modules.

Encouraging reuse. Module reuse specifications (non-implementation-specific descriptions of a module) define common functionality in ways that naturally allow software reuse without referring to any particular form (implementation) of the module.

Past efforts to increase software reuse have focused on developers. Traditionally, software developers tried to match application requirements to existing software. But matching requirements with functionality (the range of functions that a piece of software can handle) is difficult and does not encourage reuse. When systems engineers are educated about reusable module specification, however, requirements can be written to encourage reuse.

If the component specifications satisfy the needs of the application, the systems engineer simply refers to the specifications as the requirements. For example, if an application's error-logging needs were satisfied by an available error logger, the requirements would be defined by the error-logger specification.

Incremental integration. With little or no reuse, much development effort is usually required to provide an *executable* version of an application. As a result, the executable version is available so late in the development process that only minor changes in the application can be accommodated. In addition, integration is done in large steps that bring together sizable pieces of untried software that take considerable time to test.

On the other hand, with a significant portion of the application coming from integrated reusable modules, developers can create an early executable version of the application. They can add functionality in increments to preserve the integrity of the application and minimize the test interval.

Having an early executable version of the application provides timely insights into performance, gives greater visibility of missing functionality, and makes it easy to demonstrate software to customers and obtain early feedback on requirements.

Library of Reusable Modules

AT&T maintains a library of integrated reusable modules for transaction-oriented applications written on the UNIX operating system (System V, Release 2, or later). The library requires that applications use:

- Certain rules for application component design and interprocess communication
- Standard interfaces and protocols
- Standard component implementations.

Rules. The library of integrated modules requires that applications follow a client/server model. In this model, a relatively large number of client processes (typically associated with users) request and receive processing from a small number of server processes (associated with a particular application task).

The library software that manages the client/server model is based on the transaction-processing-monitor component of the AT&T Tuxedo transaction processing system, a component referred to as System /T.

System /T uses an application decomposition model in which communication is by standard UNIX System V message passing. System /T features a routing mechanism known as a *bulletin board*—a "traffic cop" that directs the flow of messages between processes. The bulletin board resides in globally addressable memory. Processes access "directions" there and follow them according to standard routines included in each process.

Another Tuxedo system component—the *field manipulation language* (FML)—constructs and manipulates messages between processes. FML data structures are self-defining; each field carries a code that indicates its identity and, for variable-length fields, its length. Communicating processes do not have to be informed, in advance, of the exact layout of the message, and they do not have to be recompiled if a message layout changes. Thus, FML makes communication flexible and data-independent, which leads to easier reuse of application code.

Interfaces and Protocols. Standard, well-defined interfaces and protocols for components are a key element of module reuse. For each component, an interface and protocol definition describes fully the syntax and semantics of all communications, whether to or from other components, test devices/components, performance-monitoring software, or controlling software. With syntax and semantics fully defined, developers can interface application code readily.

Another important benefit results from the definitions. Suppose an application can be 80 percent satisfied with standard components, but requires a completely new event/error logger (for example, if errors and events must be retained indefinitely for legal reasons). Knowing precisely how this nonstandard component must interact with other components, the application developer can produce a new event/error logger that is compatible with the other standard components and meets the needs of the application.

New components like this, with standard interfaces and protocols, can be collected in a catalog. For later applications, developers then will have a choice of one or more implementations and know that their selection will meet the communication needs of other components.

Several interfaces and protocols have been defined. One is Standard Query Language (SQL) for databases.[2] If an application uses SQL as the interface for its database, it can use any of several alternative database managers, chosen for its special features and performance characteristics. For example, the Tuxedo System /D, the Oracle® system (registered trademark of Oracle Corp.), or the Informix® system (registered trademark of Informix Technologies, Inc.) can be selected for the database manager.

Other defined interfaces are that to a terminal based on asynchronous characters (defined by the standard UNIX system packages `curses/terminfo`) and that for application messages between processors [defined by protocol X.409[3] of the International Telegraph and Telephone Consultative Committee (CCITT)].

Components. The module reuse library offers several components, some of them available in alternative implementations. Examples are described below.

Data Dictionary. The Data Dictionary[4] component is the repository and source of application-specific database information. (Other components either are database-independent or derive their database knowledge from the Data Dictionary). It is important because it lets developers refer to database-specific information in a consistent way.

The Data Dictionary manages the *vocabulary*—the set of data definitions peculiar to an application. A data definition is a description of a data item, such as its name and its data type (short, floating point number, string, for example). In addition, a definition may contain syntactical checks for the data item, such as requiring that its value be a number between 1 and 10 or that it begin with a capital letter and be no longer than 18 characters.

The Data Dictionary also manages the *grammar* of the application. Like the grammar of the English language, the grammar in the Data Dictionary states the rules for combining data items (words) to produce database relations (sentences). The rules indicate the required data items and those that are optional for a given relation.

User Interface. There are several alternative User Interface components. We describe one alternative here—a component that combines an easy-to-use, yet powerful, Interface Manager with a Visual Editor that helps developers set up the interface.

The Interface Manager controls input and output at each user's terminal. It presents a menu on the terminal screen from which a user either can select forms for entering data and displaying data or can proceed to further menus. Because menus act as a map to forms, they are ideal for inexperienced users. More proficient users, on the other hand, can bypass the menu-selection mechanism by using a command line to move to any form or menu in the application.

A data-entry/data-display form may consist of several pages, each with several fields in which the user supplies data and the application software responds with output data. Examples of fields are those for text, date and time, numeric data, and selectable lists. When a user enters data in a field, the Interface Manager automatically checks it for syntax and alerts the user if an error has been made.

In addition to fields, forms may contain *literals*—fixed text items, such as labels, titles, column headings, and instructions, that can be placed anywhere on the form.

The Interface Manager contains many features that make it convenient and efficient to use. The *selection from a list* feature, for example, is a simple but effective mechanism for entering data into a field on a form when the choice is a fixed set of items, large or small. The Interface Manager offers on-line help, windows for displaying information of short-term interest, field-display characteristics that can be changed to suit the application, built-in text editing, and scrolling. In addition, it lets the user reprogram the keyboard so that a single keystroke will summon functions such as help, scrolling, and text editing.

Although the Interface Manager was designed primarily for the AT&T work group system (WGS) product line, it also supports other terminals compatible with ASCII (American Standard Code for Information Interchange) data format. The AT&T WGS's local computing power allows user-interface processing to be done on it instead of on a host processor.

The Visual Editor part of the User Interface *instantiates* the user interface—that is, it produces from an abstractly defined concept a concrete code that can function in a real environment. It is a "smart" editor with close links to the Interface Manager and the Data Dictionary and can be used to change definitions in the latter component.

The Visual Editor shows the user forms and menus on the screen as they are being edited. The user positions the cursor on the screen to declare where fields, labels, and menu choices should appear. A compiler then changes the form or menu definition into a format that can be used efficiently by the Interface Manager. Because the Visual Editor and its compiler automatically reference the Data Dictionary, it is impossible to create a form that is inconsistent with the application's database vocabulary and grammar.

Database Manager. The Tuxedo System /D is a high-performance database management system for on-line applications. With it, a user can back up or reorganize the database without making it temporarily unavailable. Its journal and recovery facilities permit quick restoration of a database lost because of a disk failure. Finally, its concurrency-control strategy gives many users simultaneous access to the database.

System /D meets the emerging industry-standard query language SQL, which furnishes a data set at a time. At the same time, it offers Record Manager, a higher-performance interface that furnishes a data record at a time. With this unique combination, System /D accommodates extra-fast processing with Record Manager and external communication and less time-critical processing with SQL.

System /D supports three types of database records:

- *Structured records* with a fixed number of fixed-length fields—a structure typical of many database systems.
- *Unfielded records* of variable length whose structure is determined and managed by the application software.
- *FML records* of variable length whose physical layout is unknown to the application software, but instead is managed by the System /D database manager. This is the same structure used in System /T. Fields in FML records are accessed through a set of FML function calls.

Network Interface. Networking, in which applications are distributed over a network of processors, has become an essential feature of many UNIX system applications. Accordingly, the module reuse library includes a Network Interface—a package of software with which a distributed application can communicate easily and securely.

The Network Interface currently supports the industry-standard transmission control protocol/internet protocol (TCP/IP), with the Ethernet® local-area network (registered trademark of Xerox Corp.) at the lower levels of the International Standards Organization's data communication model. It uses the CCITT's proposed international standard X.409 and X.410 message format as the network data format. It presents a high-level interface to the application software, so that changes to application code are unnecessary if network protocols change.

Because not all applications have the same networking needs, the Network Interface provides four classes of network transactions: interactive (synchronous); spooled (asynchronous); interactive, changing to spooled if synchronous transmission facilities are not available; and ordered interactive/spooled, in which the user gives a preferred order of transmission that is followed even if synchronous facilities are unavailable. In addition, the Network Interface offers a "multicast" option by which an application can send a transaction to many destinations with a single request.

The Network Interface has three subcomponents. A Network Client sends a transaction from an originating host (client) to a receiving host. A Network Server routes an incoming transaction to the appropriate service on the receiving host. A Network Spooler ensures asynchronous transmission to a receiving host.

The Network Interface offers a distributed bulletin board—an important extension to the Tuxedo System /T bulletin board. With this feature, services on other processors in the network can be accessed like locally offered services. However, instead of being routed to a server on the local host, a service request is sent to the host's network client, which forwards the request to the appropriate processor on the network.

Event/Error Logger. The Event/Error Logger is based on the Productivity Improvement System for Manufacturing (PRISM) application. It provides a single point for collecting, distributing, and logging event/error messages for components of the application.

Three files—all outside application processes—define error and event messages. The files contain short-, long- and extended-format versions of each message. The bulletin board routes event/error messages from the application processes to the logger.

The logger can handle messages from multiple, concurrent processes and distribute them to multiple destinations, including remote machines. The application completely controls the destination of each message.

In addition to logging and distributing, the Event/Error Logger lets users browse through its messages. Users can ask to see messages that are logged in a database through an SQL interface.

Productivity Potential. Each reusable component potentially can furnish considerable improvements in productivity. With the help of component specifications, systems engineers can state the functionality of common elements of an application more precisely and developers can find implementations that meet the requirements. In a typical application, the database, user interface, network interface, and event/error logging requirements are likely to be satisfied by reusable components or by natural extensions of them. Developers save effort in both developing and testing components.

Reusable modules have already been applied to two projects that, although of modest size, illustrate the benefits of reuse. For both projects, designers adopted the bulletin board, `curses/terminfo` interface, and SQL; they did not use the X.409 interface because they either had to meet existing interfaces or did not need networking. In both cases, they adopted the Tuxedo System /D database and PRISM event/error handler. They chose different user interfaces—a library component for one project and, to satisfy local needs, a nonlibrary component for the other.

Both projects were completed with about half the staff effort that would have been needed without reuse. This savings estimate is based on the original development cost of a reused component, if known, and on the size of a component if development cost is unknown.

In both projects, the developers decided to adhere to their original schedules instead of the shortened lead time that reuse would allow. They therefore were able to build added functionality into the first release of their software.

Looking Forward

As module reuse matures, new components, and enhancements to existing ones, will be added to the library of modules. For example, user interfaces for graphics and additional database managers will be needed. Furthermore, new components will be required for applications that are not transaction-oriented.

References

1. T. J. Dwyer, "TUXEDO™ Transaction Processing for 3B Computers," *AT&T Technology*, Vol. 3, No. 1, 1988, pp. 40-45.
2. American National Standards Institute, *American National Standard for Information Systems Database Language—SQL*, ANSI X.3.135-1986.
3. International Telegraph and Telephone Consultative Committee (CCITT), VIIIth Plenary Assembly, Document 66, Study Group VII, Report R38.
4. S. M. Huff et al., "A Medical Data Dictionary for Decision Support Applications," *Proceedings of the Eleventh Annual Symposium on Computer Applications in Medical Care*, November 1987, IEEE Computer Society Press, New York.

Biographies (continued)

Ohio State University. He joined AT&T in 1967. ***Thomas E. Buonanno*** *is supervisor of the Reuse Technology Group in the Software Architecture Department at Bell Laboratories in Liberty Corner, New Jersey. His group focuses on ways in which software reuse can improve productivity and quality in software applications. He has a B.S. in mathematics from the University of Rhode Island and an M.S. in industrial engineering/operations research from Columbia University. He joined AT&T in 1979.*

(Manuscript received April 26, 1988)