Abstract

This paper describes the experience of a large bank in designing and implementing an information systems strategy that is based on the concept of reusability. The design and implementation was performed in two stages: (1) building a prototype to investigate the feasibility and attractiveness of reusability concept for the bank; and (2) its subsequent implementation using a library of reusable entities and a programmer’s workbench. The implementation experience confirmed that applying the reusability concept to all stages of the system’s life cycle results in both strategic (e.g., improving programmer productivity and increasing the bank’s capacity for timely response to market opportunities) and operational (e.g., reducing and controlling system development and maintenance costs) benefits. It is estimated that the library of reusable entities embedded within the programmer workbench saved the bank over $1.5 million in development costs in 1989 alone. Two of the most important lessons learned in implementing the reusability-based strategy are: (1) reusability comes in many flavors and should be applied to all stages of systems life cycle; and (2) major challenges in implementing the reusability-based strategy are managerial, not technical.

Keywords: Software reusability, programmer productivity, programmer workbench, banking systems, information systems development, information systems strategy

ACM Categories: D.2, D.2.2, D.2.6, D.2.9, D.2.m

Introduction

Information systems are the backbone of the products and services offered by financial institutions and hence are viewed as important ingredients of their success. Information system trends over the last two decades point to the development of application software as a particularly critical task facing the managers of information systems. A review of these trends reveals that hardware and communications technologies have made tremendous strides in terms of increased processing power, reduced size, and lower prices. For example, productivity in semiconductors and communications industries has increased at the rates of 65 percent and 100 percent per year, respectively, over the last two decades. However, the technology of software development, displaying a productivity growth rate of only 5 percent per year, has not kept pace and is recognized as being the primary bottleneck in creating appropriate information systems capability (Gruman, 1988). Considering the magnitude of the software development costs—estimated at $125 billion in the USA in 1990 (Boehm, 1988)—the criticality of software productivity growth assumes even greater importance.

The concept of reusability is seen as an important approach for improving software productivity and has received a great deal of attention in the software engineering literature. The main theme conveyed in research literature is that the concept of reusability is certainly feasible and has great potential for delivering substantial benefits. One of the advantages of reusing software is that
Strategy for Software Reusability

well-used modules have tended to be thoroughly tested and very rarely give rise to errors (Ince, 1988). In addition, software reusability is believed to lead to improved maintainability, quality, portability, and software productivity (Bassett, 1987; Wong, 1986).

However, while reusability is a strategy of great promise, its promise has been largely unfulfilled (Biggerstaff and Richter, 1987; Parker and Hendley, 1988). The main inhibiting factors have been the absence of a clear reusability strategy (Biggerstaff and Richter, 1987) and the lack of specific top-management support, which can lead to resistance from project managers and programmers (Tracz, 1987). This resistance to implementing reusability is possibly due to the view held by some project managers and developers that reusability will lead to reduction in their budget and staff (Wong, 1986).

The reusability concept is not being implemented to the degree that it deserves because of several other reasons (Ramamoorthy, et al., 1988; Woodfield, et al., 1987):

1. Software systems are often not initially designed for software reusability. This makes the task of identifying the reusable components within the existing code extremely difficult.

2. Integrating multiple components in one application is difficult because of a large number of bugs that occur when there are interface problems.

3. With deadlines near, the priority of programmers is to fix bugs in programs quickly, making the programmers less amenable to using reusable components.

4. Investment needed to develop software reuse systems, to develop tools and methods to support both the creation and use of component libraries, and to train users is an inhibiting factor.

A generally accepted methodology for creating and implementing reusability is still lacking (Lenz, et al., 1987). An asset-based system development methodology has been recently proposed by Karimi (1990). The proposed method required integration of data and process modeling through the use of semantic modeling techniques and tools. This method emphasizes reusability at the design rather than at the code level. It is interesting to note that the reusability project described in our paper independently arrived at, and used during the prototype development, a methodology that is, in essence, equivalent to the asset-based methodology.

Practical experience with reuse has been scant, and very little has been reported in the literature. Few reusability systems that are in widespread use are based upon numerical computations and subroutine libraries for input/output and string manipulation (Horowitz and Munson, 1984; Shriver, 1987). The software productivity system at TRW has promoted software reuse and tool integration (Wartik and Panedo, 1986). An interesting result of a study of reusability at Raytheon Company showed that 40 to 60 percent of the program code was repeated in more than one application (Frank, 1981). Most studies of software reuse agree that about one-half of code from one application is reusable in another (Biggerstaff and Perlis, 1984). An early example of implementing reusability can be seen in Japanese software factories, where the benefits have justified the cost of developing reusable parts (Matsubara, et al., 1981). GTE data services used several incentives to encourage reuse—a quarterly newsletter describing its benefits, a $50 reward for programmers whose components were reused within a year, and making successful reuse a part of the project manager’s job evaluations and a condition of pay raises.

This paper describes the experience of a large bank, well known for its information systems capability, in implementing a reusability-based strategy for development of information systems. First, the bank’s rationale in choosing the reusability strategy is discussed. Then, the implementation strategy is described in two stages: (1) building a prototype to investigate the feasibility and attractiveness of the reusability concept for the bank, and (2) its subsequent implementation using a library of reusable entities and a programmer’s workbench. Finally, the lessons the bank learned in implementing software reusability are presented and future directions are suggested.

Information Systems Strategy of a Bank

The banking industry has undergone a period of turbulence in the last decade that promises to
persist in the foreseeable future (Sametz, 1984). The major changes affecting the industry have been in the areas of government regulation, technology changes, and consumer preferences.

Deregulation of the banking industry has increased the scope of permissible activities for financial and non-financial institutions, has relaxed geographic restrictions, and has gradually phased out the ceilings on interest rates payable on deposits. All these changes have increased the intensity of competition by expanding the range of services being provided by banks and other financial institutions.

Technological advances in both the computer and telecommunications industry have improved the economies of scale and scope and are having a profound impact on the functioning of a bank. Technology has given banks an ability to handle a large volume of transactions at a declining average unit cost, to support geographically dispersed operations through distributed processing, and to offer new products and delivery channels, such as automated teller machines (ATMs), bank-by-phone, automated clearinghouse, etc.

Customers' preferences and attitudes have also changed. Customers are more prone to shopping for the optimum balance of risk and return on their deposits, and are becoming more discerning in their evaluation of services. This has added to the competitive pressures on the industry.

These changes have put pressure on profit margins in the banking industry, and cost containment in all areas, including information systems, is becoming strategically important. Competitive forces have also led banks to change their product development activities. The ability to offer new products quickly to customers is becoming more critical, and the movement toward technology-based differentiation of products and delivery channels is becoming more pronounced.

Timely introduction of products requires that the corresponding applications be developed in a like manner. It is also necessary that this application development be carried out without disrupting the systems supporting existing products. At the same time, new software development and maintenance activities are also becoming increasingly more expensive. Senior information systems management is, therefore, becoming highly sensitized to the need for new ways to improve the efficiency of software development and maintenance.

**Critical success factors**

Recognition of the above factors in the early 1980s led senior information systems management at the bank to identify the following critical success factors for application systems:

1. The cost of developing and maintaining software must be reduced.
2. New applications must be developed in a timely manner.
3. The application systems as a whole must be flexible enough to evolve in a controlled manner.

Investigation into the various options available to realize these factors led to the concept of the reusability of code aided by software engineering tools, including formal methodologies for structured analysis and design of code. A review of the bank's products and services had previously revealed that they were based on a small set of basic transactions, such as open account, accrue interest, etc. Therefore, if a set of reusable programs could be created to capture these basic transactions, it was conjectured that it would be possible to recreate the significant portion of a bank's applications systems over time through the reusable software.

Use of a library of reusable software could save considerable development costs and improve the quality of software and maintenance of software products. Development costs would be reduced because some software would not have to be developed. Reusable software would also allow timely development of applications. Being a one-time development activity, it would be easier to ensure that the reusable modules belonging to the library were of extremely high quality. Because new software would be composed of these high-quality modules, the final quality of the software would be high, and this would mean lower maintenance costs over the life of a system. The maintenance and enhancement of applications could also become easier because, in many cases, maintenance would need to be performed only on a single copy of the reusable component. By allowing better system maintenance and
Strategy for Software Reusability

enhancement, the reusable code could improve flexibility of application systems. Thus, it was felt that reusability of software would satisfy all the critical success factors.

In 1982 an information system strategy based on reusability of software was proposed, and subsequently, an 18-month research and prototyping effort to examine and confirm the feasibility and attractiveness of the proposed strategy was commissioned by senior information systems management. A project team was then assembled to work on this project.

Prototype Project: Objectives, Methodology and Results

The project team began its work on the prototype by addressing the following major research questions:

1. What are the main characteristics that determine the applicability, or non-applicability, of a software reusability concept to a given application system?
2. How does one identify the reusable components of code from existing application systems?
3. Is it feasible to recreate multiple applications from reusable software, and what is the extent of reusability that can be achieved?

The concepts developed and results of the work performed by the team are described below in three subsections that parallel these research questions.

Reuse of code: Manufacturing approach to software development

To address the first question, the project team developed a concept that it labelled a "manufacturing approach" to software development. This approach involved viewing application development activities as a production operation, with completed application software as the product. This viewpoint suggests the use of a simple, yet powerful, operations management concept of the matching of products and processes (Hayes and Wheelwright, 1979). The concept recommends using a job shop-type process for producing customized, one-of-a-kind products, while using a component assembly line approach for producing standardized, large-volume products.

Within the context of application development, program types (i.e., products) can be organized along a continuum from decision support systems (DSS), to management information systems (MIS) to transaction processing systems (TPS). The development approaches can be arranged from a craft approach to a manufacturing approach. The craft approach implies an unstructured activity in which a few skilled analysts/programmers assume responsibility for developing the system. The manufacturing approach implies piecing together applications from previously constructed reusable components. Figure 1 illustrates the suggested matching of development approaches and program types. The craft approach is well-suited for one-of-a-kind situations such as those found in decision support applications. In contrast, the manufacturing approach, or the reusability of software approach, would make sense in a high-volume, transaction processing type of banking application (e.g., demand deposit, credit card, installment loan, etc.) where a good deal of commonality exists among applications.

Development and use of the manufacturing approach concept helped identify the appropriate development approach for programs in each application system. The programs that could be implemented under the manufacturing approach were selected and targeted for further analysis in order to identify reusable components. These components, representing commonly used processing steps within a particular group of applications, would thus act as a set of "building blocks."

A prerequisite to choosing reusable building blocks is that the set of targeted applications should display a sufficient degree of commonality. This requires that transactions involved in underlying products or services also have a sufficient degree of commonality. For example, deposit products such as savings accounts and

---

1 In the banking industry, application systems are organized according to the products they support. Each application typically consists of several programs that individually perform the different tasks necessary to support a product. Thus, an application system consists of programs of various types, ranging from TPS to DSS.
money market accounts involve similar transactions. Our observation was that this condition was met to a larger degree by the retail banking business. The wholesale and international banking and trust business had products that each displayed a substantial amount of uniqueness. It was, therefore, concluded that the concept of reusability, at a business function level, was applicable more on the retail banking side.

Methodology for identifying reusable components

Identifying reusable components was the first critical task in the prototype effort. The project team used both analytical and empirical methods in parallel to accomplish this.

The empirical method of the study involved examination of the bank's processing needs by interviews with system users and by a detailed review of current application systems. Although the project team studied all banking products and application systems, it concentrated its efforts on those products and programs that were considered to be most appropriate for the application of reusability. The interviews and application reviews were followed by an exercise to identify parts of programs that were used in more than one application and were based on similar processing logic.

The analytical method of study involved examining the fundamental relationships between a financial institution and its customers, and describing the basic financial services and supporting transactions associated with these relationships. Based on this examination, the reusable components required to support the common transactions were designed. We observed that a degree of commonality existed within the group of deposit products and within the group of loan products.

Two kinds of reusable components were identified at the banking function level: monetary and non-monetary. Monetary components addressed all monetary transactions that by definition have an impact on account balances (e.g., deposit to account, interest accrual, etc.). Non-monetary components addressed all other important transactions performed in a bank (e.g., open account, renew account, etc.).

In addition to the banking industry's specific components, the team also identified components related to the general data processing functions (e.g., sorting, database access, etc.) or to transaction management (e.g., creating transaction log, maintaining control totals, etc.). We found that developing reusable components of this type hinged on the standardization of data.

Figure 2 shows examples of relationships between the account types (or bank's products) and reusable components. The "renew account" component is relevant to certificates of deposit but does not apply to checking and savings accounts. Similarly, interest must be calculated for savings and certificates of deposit, whereas it need not be calculated for an interest-free checking account.

The analytical and empirical methods proved complementary to each other. The analytical method provided a conceptual framework for organizing and identifying the components, whereas the empirical method supplied the details necessary in specifying components in terms of the data elements and the processing logic used.
Strategy for Software Reusability

<table>
<thead>
<tr>
<th>Products</th>
<th>Monetary</th>
<th>Non-Monetary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Deposit to Account</td>
<td>Calculate Interest</td>
</tr>
<tr>
<td>Interest-free checking</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Savings</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Certificates of Deposit</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Figure 2. Reusable Components and Banking Products

Prototype development

The end result of the analytical and empirical work was the identification and specification of reusable components. An entity-relationship-based enterprise data model was also developed for the bank. A high-level design of all transaction processing systems was created using this data model and the data flow methodology. A prototype of a selected subset of the bank's product was then constructed using a highly productive fourth generation language.

In the prototype development effort, about 30 reusable components were identified and developed. Each component was found to be reused, on average, in approximately eight different products. These reusable components accounted for an estimated 60 percent of the software needed to construct an existing or new banking product. The study also found that the components under the monetary category could be reused in their entirety across different products. These components provided the highest form of reusability that could be achieved. Non-monetary transaction components shared processing logic, although they were not always identical in terms of the data elements used. Developing reusable components for the non-monetary transactions, therefore, hinged on resolving the differences in naming and formatting the underlying data elements and in creating a unified enterprise data model.

In summary, the prototype validated the feasibility of the software reusability approach. It also confirmed that the use of a standardized reusable code would result in both strategic benefits (e.g., increasing the bank's capability for timely response to market opportunities) and operational benefits (e.g., reducing and controlling system development and maintenance costs).

Experience in Implementing the Software Reusability Concept

Subsequent to the completion of the prototype in mid-1984 and its successful presentation and demonstration, senior information systems management decided to implement the proposed concepts and created a small organization for that purpose. This organization was given an explicit charter to improve programmer productivity through reusable software, and one of the first projects undertaken by the unit was the establishment of a reusable entity library (REL). In approving the REL project a "sunset" clause was set. The clause stated that the success of the project would be measured in terms of the actual usage of the REL entities. If, after a reasonable time, usage did not match expectations, the effort would be redirected or curtailed. This library was made available in early 1986 and was later enhanced to create a standard application development environment, termed programmer workbench, in 1988. These two projects are described in the next subsections.
Phase 1: The reusable entity library

Very early in the REL project, the team realized that the biggest obstacles, and yet prerequisites for success, were successfully persuading managers at all levels and training programmers to think in terms of reusable programming constructs. These two tasks were not easy but were accomplished through multiple activities.

The benefits of reusability are not immediate for any programming team working under deadlines and coping with day-to-day operational problems. In fact, most project managers tend to focus on upfront learning costs, which may lead to productivity loss in the short term and cause potential delays, rather than on reduced development cost and time and on the downstream benefits of higher-quality systems that have reduced maintenance costs.

The banking function-level reusable entities were already identified and specified as part of the prototype effort. It would have been feasible to create a library of entities based on these specifications, but developing a complete retail banking application such as a demand deposit system from scratch, using these entities was simply out of the question. These systems represent an investment of literally hundreds of millions of dollars, and although the current systems may not be perfect, with regular maintenance they seem to work satisfactorily. Therefore, to recreate existing applications just to make use of reusable components was clearly risky and unjustified.

A study of programmer activities at the bank indicated that a typical programmer spent about 25 percent of available time on new application development, about 60 percent on maintenance and enhancement of old programs, and about 15 percent on user support and operations. Hence, limiting the application of the reusability concept to software alone was deemed short-sighted because it would have addressed only about one quarter of a programmer’s activities. Therefore in defining the REL project, a broad definition of reusable entity was adopted. This definition included any software or procedure that could be used by system developers in all stages of the system life cycle. Examples of reusable entities include subprograms, utilities, parameter-driven CLISTS, JCL, file definitions, etc. The design of REL addressed both the technical and manager-
Management Reporting

The major purpose of this function was to set up a process for tracking the usage of entities by programming areas. Quarterly usage reports were produced to help senior management monitor the project in terms of the economic benefits achieved. These reports were also used as a basis for an incentive scheme that encouraged programmers to use the REL. The scheme was based on recognizing and rewarding the teams that made above-average use of the REL.

The first version of the REL design was in place by the end of 1985. The initial release of the library in early 1986 consisted of approximately 20 entities. The number of entities grew to 40 by the year end. Over time, the entity usage grew steadily from approximately 200 usages in mid-1986 to over 3,000 usages per month by the end of 1987.

Although the initial objectives of the reusability project during the prototype phase were primarily to identify and implement reusable components in functionally similar banking applications, we found that programmer demand and the big payoffs were in developing common utilities that helped programmers alleviate their daily operational "bottlenecks." The requests received from technical representatives dealt with managing code in a multi-site environment, handling turnover procedures, interfacing with core systems (e.g., general ledger, output services, etc.), automatically enforcing programming standards, and improving programming efficiency.

The team estimated the benefit potential of each entity by computing the dollar value of the time saved per entity usage. As expected, a wide distribution of benefits was observed. For example, only a few dollars were saved by using an entity that determined the optimal blocking factor for a particular I/O medium, but the use of an entity for interfacing a purchased package with the bank’s standard printing utility resulted in savings of thousands of dollars.

The library achieved the break-even volume of operation in the third quarter of 1987 and met the usage expectations of senior management. It was estimated that the library contributed about a quarter of a million dollars in benefits in 1987. In terms of the manpower used, about five man-years were spent developing the REL, with half that amount spent in 1987 alone.

Phase II: The programmer workbench

The analysis of the entity usage of REL showed that the increase in usage was directly linked to the number of entities in the library. In late 1987, usage had levelled off in the range of 3,000 to 4,000 per month. The main reason for this leveling off, determined through informal programmer feedback, was that the information utilities supporting the library shell were fast becoming inadequate for the task of presenting the contents of the library to the programmer community in an increasingly complex application development environment. The bank had recently introduced a new relational DBMS and a new teleprocessing monitor, and had reorganized its development activities into three separate sites. Thus, as the introduction of additional entities occurred, the cost of finding, understanding, and using entities had increased to a point where it seemed to have imposed a negative influence on programmer productivity.

The team’s solution was to develop a programmer's workbench to provide a single consistent application development environment for all programmers. The workbench would provide a consistent framework for housing all reusable entities. All screens, help facilities, and interfaces were made consistent across all entities, and thus the cost of learning the features of a new entity and operating in a new environment were minimized. The workbench shielded the programmer as much as possible from the idiosyncrasies of the actual operating environment. The programmer would only select the actual entity of interest, and the workbench would create the necessary conditions (in terms of required JCL, job class parameters, etc.) for the entity to work properly in the current operating environment. The workbench design used a hierarchy of menus for beginning users, with the added capability for experienced users to access desired functions directly through the use of function keys.

The programmer workbench project was started in the third quarter of 1987 and was implemented in the second quarter of 1988. The initial release raised the REL usage from about 3,000 to 6,000 per month. Since then, the functionality and contents of the workbench have been continuously enhanced; today the REL usage is over 20,000 per month, and about 900 distinct users log on
to it. A graph depicting the increase in usage over the last four years is shown in Figure 3. A recent benefit calculation showed that the programmer workbench has saved the bank over $1.5 million in development costs in 1989.

Lessons Learned in Implementing Software Reusability

The workbench/reusability effort is an ongoing project at the bank. The current scope of the project is somewhat broader than the one envisioned during the project startup. Nevertheless, the experience gained in implementing software reusability during the last eight years has taught several valuable lessons. This section discusses our recommendations for implementing reusability.

Reusability improves productivity and cost performance

Based on our experience in implementing reusability at the bank, the strategic and operational benefits of reusability are clear. Reuse of standard, well-tested, off-the-shelf software components improves programmer productivity and saves significant cost in developing software. The high quality of reusable components also reduces application maintenance in the long run. In terms of the investment costs of the reusability strategy, the bank has spent little more than $1 million over the last five years. In comparison, the benefits realized during the same time period, in terms of avoided development costs, are more than $2 million, with an annual savings of approximately $1.5 million in 1989 alone. Thus, from a cost containment viewpoint, implementing software reusability has proven to be an attractive proposition.

With reduction in time required for software design, development, testing, and implementation the overall software productivity has also improved. The case for strategic benefits of enabling quicker introduction of new products is, in our opinion, also quite strong. But, not having concrete and quantifiable proof, this benefit should be viewed as a strongly plausible hypothesis that remains to be proven.

Reusability comes in many flavors

One of the main lessons the team learned is that, from a broad perspective, reusability should ideally be viewed as a productivity enhancement...
tool. The low growth in software productivity and the general shortage of quality programming resources are proving to be the main bottlenecks in using information systems in the bank. We feel that promoting reusability, in its broadest sense, is an important tool for alleviating that bottleneck.

Traditionally, the work on reusability has focused on isolating functional commonality across application systems. This viewpoint mainly addresses the coding aspects of a programmer's job. However, coding is not the only task a programmer performs. The process of software development and maintenance involves a number of fairly standard processes that must be repeated for each project. Some of these typical programmer activities include business modelling, data analysis, conforming to standards, testing, and turnover procedures. Wherever possible, these activities should be considered as candidates for the application of reusability. After all, from a business perspective, the management is interested in developing a high quality system that is delivered on time and within budget.

At the bank, reusable entities have been classified into five groups. We view the banking function-level software reusability as a high-level one, whereas the reusable entities that deal with operational/informational aspects of a programmer's job are viewed as the lower-level ones. The entity groups listed below are arranged from a high to low level in terms of standard system life cycles—development, testing, implementation.

**Banking Function Entities:** This group consists of common banking functions such as interest calculation routines, date routines, etc.

**Application Development Entities:** These entities help software development and include utilities for manipulation and scanning of data and files, source code development, code restructuring and quality analysis, and screen generation. They create standard interfaces to accounting and reporting systems (such as general ledger) and to various DP processes (such as input/output, teleprocessing monitors, etc.).

**Application Testing and Maintenance Entities:** These relate to activities such as online and batch testing, test data generation, scripting, and flow pathing.

**Operations and Standards Entities:** These activities deal with responsibilities such as conforming to software standards, requests for computing resources, generation of run books for scheduling dependencies of programs, and procedures for a program's turnover to operations.

**Application Management Entities:** These include traditional project management, resource allocation, and status reporting activities associated with software development. Also included in this group are activities important to operations function such as monitoring the on-call responsibility.

**Information Utilities:** These deal with information required by software developers and include items such as manuals, entity information, training, bulletin boards, and hotlines.

Our hypothesis is that the benefit per usage of high-level reusable entities is higher than the low-level entities. But it should be noted that the high-level entities are more difficult to identify and develop, are likely to be used with lower frequency, and are, in general, more difficult to implement. The lower-level entities, on the other hand, give smaller benefits per usage, but are easier to develop and implement. Lower-level entities are also used with higher frequency and are more easily accepted by programmers. Hence, in implementing reusability, it is advisable to implement lower-level entities in the beginning.

Another difference between higher-level entities and lower-level entities is that the value of the former is related to the size (in lines of code) of the module; however, the value of the latter is determined primarily by the time it saves the programmer in completing a particular procedure, not necessarily by the size of the reusable entity.

The major challenges are not technical but managerial

Persuading managers and programmers to buy into the concept of reusability involves a major change from traditional practices. This change is not likely to be automatic, and we feel that the entire exercise is more likely to fail from a lack of management commitment and programmer interest than from any technical reasons.
Securing senior management's approval is relatively easy. Because of their long-term, strategic viewpoint, the benefits of reusability are evident to them. But middle-level managers, including direct managers of programmers, are more resistant to the concept. The difficulty in obtaining buy-in is possibly due to their short-term planning horizon and their performance evaluation, which emphasize meeting the target completion dates and budgeted costs. These factors make them reluctant to accept any new process that implies a certain amount of up-front learning costs for their programmers and a possible delay in project completion. Thus, from an implementation viewpoint, it is important that these managers be specifically kept in mind in marketing the reusability concepts through such means as presentations, seminars, and senior management memos.

Getting programmers to use the reusable software is, of course, the most important prerequisite for success. As we found in our experience with the programmer workbench, meeting the information requirement of programmers and making reusable components an integral part of a standard software development environment is key to the success of reusability.

Implementing reusability: Other considerations

In implementing reusability, it is important that the applications targeted for potential use of reusable software are transaction processing types. In the bank's case, it was found that this reusable software was more applicable to the simple, retail banking products as opposed to the more complex wholesale banking or trust products.

As discussed earlier, it is advisable to implement lower-level entities in the beginning of a reusability project. Programmers will then immediately realize that reusability helps them solve their operational problems and improve their productivity. An important consideration for improving programmers' usage of reusable entities is to provide a monetary or other suitable incentives.

As in any project dealing with new concepts, achieving a few success stories early in the game is very important, and starting with the lower-level reusable entities is a good way to ensure that. The high-risk, high-payoff entities in the form of reusable software modules can always be implemented once the programmers and their managers have bought into the concept of reusability.

Conclusions and Future Direction

This project has provided important insights in strategic and operational implementation of reusability in software development. Although the initial objectives were primarily to identify and implement reusable components in functionally similar banking applications, the project team found that programmer demand and the big payoffs were in developing reusable entities that helped programmers alleviate their daily operational "bottlenecks." Developing the programmer's workbench so it would provide a consistent software development environment for housing all reusable entities proved to be a successful approach. Initially it was difficult to get the buy-in from programmers and their direct managers, and it was essential to market the reusability concept to these managers. Achieving a few success stories early in the implementation proved to be very important, and starting with the lower-level reusable entities ensured that objective.

The present goal of the programmer workbench/reusability project is to integrate existing mainframe-based CASE tools into the platform to enhance its capability as an integrated software development environment. The project team is also excited about the developments in PC/workstation technology. Plans to position the workstation and REL to better utilize these technologies and to possibly migrate to these platforms over the next several years are currently underway.

References

Strategy for Software Reusability


About the Authors

Uday M. Apte is an associate professor of management information sciences at the Cox School of Business at Southern Methodist University, Dallas, Texas. He holds a Ph.D. from the Wharton School, University of Pennsylvania, where he has taught operations management and management information systems in the M.B.A. program. His research interests and publications are in the areas of service operations, global manufacturing, manufacturing automation, and MIS strategy in the financial services industry. He has over 10 years of managerial experience in information systems and operating divisions of banking and insurance companies. He has also served as a consultant to major corporations in the manufacturing and financial services industry. His earlier education includes a bachelors in chemical engineering from the Indian Institute of Technology, and an M.B.A. from the Asian Institute of Management.

Chetan S. Sankar is an assistant professor at the Department of Management at Auburn University, Auburn, Alabama. He has extensive industrial experience in telecommunications management. He has designed network management systems for multi-national corporations. His publications have appeared in Management Sciences, Information Management Review, Decision, IEEE Transactions on Professional Communication, and the Naval Logistics Quarterly. He holds a Ph.D. from the Wharton School, University of Pennsylvania.

Joel E. Turner is a senior vice president with the Profit Management Group in Malvern, Pennsylvania, where he directs the firm's product development efforts and systems implementation engagements in the financial services industry. Prior to joining the Profit Management Group, Mr. Turner was a senior vice president with Lit- tlewood, Shaun and Company, and also worked in strategic planning for the Information Systems Department at Mellon Bank. Mr. Turner received
an M.B.A. from the University of Michigan and a B.A. from Kenyon College.

Meru Thakur manages the High Technology Section of the Information and Systems Department of Mellon Bank and has been responsible for evaluating new products and technologies and developing measurement standards for programmer productivity. His team has been responsible for introducing the Reusable Entity Library, the programmers’ workbench and software reengineering tools and methodologies to the Bank. In 1988 he was responsible for developing the first executive information system for the Office of the Chairman. He received his Ph.D. in decision sciences from the Wharton School, University of Pennsylvania.