The business case for software reuse

by J. S. Poulin
J. M. Caruso
D. R. Hancock

To remain competitive, software development organizations must reduce cycle time and cost, while at the same time adding function and improving quality. One potential solution lies in software reuse. Because software reuse is not free, we must weigh the potential benefits against the expenditures of time and resources required to identify and integrate reusable software into products. We first introduce software reuse concepts and examine the cost-benefit trade-offs of software reuse investments. We then provide a set of metrics used by IBM to accurately reflect the effort saved by reuse. We define reuse metrics that distinguish the savings and benefits from those already gained through accepted software engineering techniques. When used with the return-on-investment (ROI) model described in this paper, these metrics can effectively establish a sound business justification for reuse and can help assess the success of organizational reuse programs.

Rapid advancements in hardware technologies have resulted in a highly competitive hardware market and an increasing requirement for software to support and exploit that hardware. Consumers demand leading-edge software as computer hardware rapidly becomes a commodity business. Only those companies that bring the latest technologies to the market quickest and at the lowest price will survive in this environment.

Successful software organizations must somehow keep pace with these hardware changes without incurring excessive cost that would result in noncompetitive pricing or reduced profit. In addition to responding to the changing hardware environment, the successful software organization must also respond to changing user expectations. With the widespread use of computers across numerous applications and user environments, the computer user has become much more knowledgeable and demanding. Customers want easy-to-use, intuitive computer systems. They expect the systems to operate reliably and to perform without noticeable delays. Competitive software organizations must invest in new designs and solutions to meet these new functional demands.

This fast-changing, highly competitive environment challenges the software organization to do the following:

- Increase productivity to support and exploit new technologies and produce more function in a shorter period of time
- Reduce development costs so as to sell both system and software at a competitive price (This requires developers to produce more with fewer resources, and to improve product maintainability and quality to reduce maintenance expenses that add cost to the product.)
- Improve software quality to meet the user's functional requirements and expectations (This requires additional design and development effort, with additional expense and time.)

Unlike other industries, software development lacks the major breakthroughs in process and methods to enable it to keep pace with the market. For example, the last major breakthrough in software productivity came with creation and proliferation of high-level programming languages.[1] Brooks states that most observers credit the progressive use of high-level languages with at least a factor of five in productivity. This productivity gain came from reducing the complexity of programs through abstract constructs such as data structures, types, and operations. Today's goals for software reuse strongly parallel the goals and achievements of high-level languages.[2] The concepts of abstract data types and hierarchical types used in object-oriented designs reduce program complexity by allowing the designer to inherit behavior or refer to existing designs.

Without further breakthroughs or significant advancements in software productivity, most software organizations find themselves in the midst of a software crisis that inhibits their ability to produce manageable, high-quality, cost-effective software.[3,4] Aging software companies cannot quickly react to hardware advances because they did not design their software to adapt to new requirements or for ease of maintenance.[5] In addition, as they make modifications to existing software to fix problems or add function, the stability and quality of their design and code begins to decline. This results in reduced quality and reliability, higher maintenance costs, and reduced responsiveness. High software development costs result in lower profit margins because of the need for competitive pricing.

A software organization faced with these challenges must make fundamental changes to remain a competitive and viable software producer in the future. What can the software manager do to meet these challenges? One hopeful solution lies in software reuse.

Software reuse has the two major benefits of improved productivity and quality,[6,7] among others. Integration of software reuse into each stage of the software development life cycle can provide long-term progress in solving productivity and quality problems and, therefore, significantly reduce the cost of developing software. Estimates for some applications show that they contain less than 15 percent of application-specific code. The remaining 85 percent comes from common, redundant, and potentially reusable software components.[8]

Great gains can come from tapping only a small part of that potential. Even if the reuse opportunity for a particular application is only 15 to 20 percent, the savings in software development time and expense can result in tremendous cost savings and competitive advantages for the software organization. For example, GTE Data Services initiated an Asset Management Program in 1986 for the purpose of creating a corporate collection of reusable assets. During its first year, GTE had a 14 percent corporate reuse rate and an overall estimated savings of $1.5 million.[9] Another example comes from IBM Hursley, where they experienced a reuse rate of 20 percent in the Customer Information Control System (CICS*) for Extended System Architecture, V4R1. Using the business case calculations described in this paper, IBM Hursley estimated a total return on investment of $2.74 million.

However, there is a cost to achieve these benefits. Reusing software requires identifying, retrieving, and integrating the software into the product. These investments in time and resources must result in benefits from reduced development cost and maintenance. Developing software for reuse by others requires additional development, validation, and support investments that must be weighed against the benefits to outside organizations and any cost recovery plans. The additional effort to produce reusable software often raises short-term product costs. The financial return is not always realized in the immediate product but in future products. In a competitive cost- and schedule-focused market, this makes it difficult for the software manager to gain financial support for long-term software reuse investments. How an organization balances its investments in software reuse with short-term product demands may mean the difference between long-term business success and failure.

This paper describes the elements of software reuse and provides a set of metrics used by IBM for analyzing the cost-benefit trade-offs of reuse investments. We define reuse metrics that distinguish the savings and benefits from those already gained through accepted software engineering techniques. We describe a model for return-on-investment (ROI) analysis that we effectively use with these metrics to establish sound business

justification for reuse and to assess the success of organizational reuse programs. We provide these metrics and analysis techniques as a tool to assist software managers and developers in determining the benefits and measuring the progress of software reuse investments at both the project and corporate levels.

Software reuse concepts

The term *software reuse* is often confused with *code reuse*. In fact, software reusability has many aspects, including code reuse. Although code reuse is perhaps the simplest and best understood aspect of software reuse, it does not represent the greatest potential benefit. Organizations spend much more time in a typical software development cycle on design, analysis, and specification than they do in the actual coding phase.

We use the term *reusable software engineering* to refer to the reuse of broad classes of software information.[10] This broad-spectrum approach to reusing existing software components includes reusing software work products such as requirements, architectures, designs, algorithms, data types, code modules, documentation, test cases, and customized tools.[11] Examples of applying software reuse at different abstraction levels[12] include:

- Reuse of designs in different software or hardware environments. Different specifications can be derived from the same design to meet system-specific requirements, such as performance constraints. The potential for reusing designs is great because they do not contain detailed implementation-level decisions.
- Reuse of specifications in different application environments. Environment-specific code, such as target hardware support, can be derived from the same specification.
- Reuse of code and test cases creating and testing different implementations of a specification. Code and test case reuse represents the most tangible type of reuse. Unfortunately, it is often difficult to find code that we can reuse without change. Implementation details often tie software to a specific programming language, hardware and system environment, particular application design, or data structure. The larger the code element, the larger the payback for reuse. However, large code elements are more likely to require modification. Smaller code segments are easier to reuse, but can be more difficult to locate.[13]

We need not limit reusable software information to design, code, and specification components. S. C. Bailin, R. H. Gattis, and W. Trusczkowski explore the reuse of design knowledge.[14] The knowledge obtained from previous efforts can apply even when we do not specifically reuse a software component. Historical data can be analyzed to project risks, resource expenditures, design trade-offs, and alternate approaches. The knowledge represented by these data can be applied to new projects as reusable experience.

The reuse metrics and investment model defined in this paper are tools for measuring and analyzing code reuse. Code reuse is the simplest form of reuse to measure, because most software organizations deal with some type of code metrics. Furthermore, code measurements not only serve as an excellent indicator of programmer productivity, they also make a good secondary indicator of effort expended in other phases of the development cycle. Until we develop a comprehensive system of measuring and evaluating reuse in each of the individual software life-cycle phases, code reuse metrics will remain our best method of assessing the overall benefits of reuse.

Software development with reuse. There are two primary approaches for building software from reusable components: the composition or building-block approach and the generative approach.

Composition approach. In the composition or building-block approach, programmers create software from existing atomic components that they usually retrieve from a reuse library.[15] The building-block approach requires components with encapsulated function, well-defined and specified interfaces, and known quality. Knowledge of the function and interfaces is sufficient to use the component. Constructing a program from building blocks requires connecting one reusable component to another. Programmers achieve this through message passing techniques or by writing glue code to link the parts together. This approach to reuse is patterned after processes used in other industries such as hardware design and gunsmithing. For example, computer hardware manufacturers design circuits by assembling reusable components and chips.[12] The gunsmithing industry has undergone an evolution from cus-

tomized, hand-built processes to industrialized processes using interchangeable parts.[3] The building-block approach to software reusability introduces these processes into the software industry.

Generative approach. In the generative approach, automated tools or generator programs operate on reusable entities. These entities may include code fragments, patterns, design information, transformation rules, and specialized languages. The Draco approach[16] is an example of a generative approach based on an analysis of the application domain and a system specification written in a domain language. Draco is an interactive tool that assists the software designer in the production of executable code from application domain objects.[17]

Reuse consumers and reuse producers. B. H. Barnes and T. B. Bollinger define a reuse producer as one who works to increase the reusability of work products and a reuse consumer as one who seeks to reduce costs through reuse of work products.[18] Reusable software engineering is concerned with both the consuming and producing of reusable software information throughout the software development life cycle. Although software reuse is most commonly associated with code reuse, reusable software information may also take other forms, as described earlier.

Consuming reusable information. The identification of potential sources of software information reuse should be performed early in the software development life cycle, as a part of the initial requirements and system domain analysis. We can reuse software parts to develop rapid prototypes to help us gather and validate user requirements. The earlier we identify a reuse candidate, the greater the potential benefit, because we can reuse designs, specifications, and documentation in addition to code modules.[19]

There are four basic steps involved in consuming reusable information:

1. Locating and accessing the information—The ease with which a programmer locates reusable parts affects the amount of time spent searching for candidate reuse information. Added time means added cost. Approaches for locating reusable information vary widely, depending on the availability and existence of tools, well-defined classification schemes, and component reuse libraries.

   The most labor-intensive search method and the informal method most commonly used by software developers consists of a manual search through existing information libraries. Even if the reusable software is well documented and readily identifies the function and environment for the reusable part, this method may have limited success and can be very tedious.

   At the other extreme, the most effective way to locate reusable information is to use sophisticated search and retrieval tools based on well-defined classification schemes. R. Prieto-Diaz and P. Freeman developed a "faceted" classification and retrieval scheme that provides search mechanisms whereby the user enters descriptive terms and initiates queries to locate candidate reuse information.[20] The user applies synonyms to locate information that is similar to the desired part, if an exact match is not available. The search for reusable information can become much more efficient and effective as search and retrieval tools of this nature become commonly available, especially for very large library systems encompassing a wide variety of software.

2. Assessing the ability to reuse the information—The ability to reuse a candidate software component depends on how closely the reuse candidate meets the reuser's requirements. It also depends on the availability and completeness of information pertaining to the candidate reusable component. W. Tracz uses an analogy that compares used cars to used software[21] and describes several factors that could influence the decision as to whether to reuse a candidate software component:

   - Does the candidate meet the needs? Buyers should have a strategy for evaluating candidates (features, performance, quality, availability, price).
   - Does the candidate meet the base requirements without adding excess function or baggage? Extra functions may not pull their weight.

- Does the candidate have a proven quality record? A use and quality history for the candidate would indicate the likelihood of future potential bugs.
- What is the candidate's maintenance record? The quality of the candidate can be evaluated by looking at the type, frequency, and severity of the problems already found.
- What is the reputation of the candidate producer? If no quality or maintenance history is available, the reputation of the provider can be used to estimate the quality of the candidate.
- Does the candidate appear to be well-structured and documented? The exterior workmanship can indicate the overall maintainability and reliability of the candidate.
- Does the candidate comply with documentation, interface design, and testing standards? If the candidate meets easily observable design specifications, it probably runs well, too.
- Who maintains the candidate after it is reused? Will assistance be provided for modifying and repairing the candidate if necessary?

3. Adapting the information—We gain the most benefit from reusing software information "as-is," without modification. As soon as we modify the reused information we incur additional costs beyond the modification, including testing the changes and maintaining the resulting software. As explained later, this paper presents metrics for measuring the costs and benefits of software reuse without modification.

   However, it may not always be cost-effective to reuse software information even if we customize the component. If a reusable component requires extensive adaptation or lacks key documents, the cost to reuse that component may become higher than if we develop an entirely new software component.

4. Integrating the information into the system—Once we obtain and possibly modify all the reusable parts, we must integrate and test them in our system. If the system is made up entirely of reusable parts, the construction or building-block approach can be used to assemble the parts. More likely, the system consists of a combination of new and reusable parts. In that case, a hybrid construction approach is necessary. Hybrid construction simply consists of combining the new and the reusable parts, possibly with the use of custom-developed "glue code."

   When the percentage of unmodified, reused code is very high, the system is similar to a system undergoing maintenance. In a maintenance environment, we build upon the existing system base of design, code, and documentation. The opportunities for reuse are in the new and modified code. In these systems, the reuse is referred to as *adaptive reuse*.[18] With adaptive reuse, we confine the new or modified code to isolated locations to minimize the impact to the overall system.

Producing reusable information. A software enterprise typically invests in producing reusable software as a means of improving its overall productivity and quality and reducing long-term costs. Reusable software information may also be developed for the purpose of obtaining revenue, licensing fees, or royalties from marketing it to other organizations. Whatever the motivation, we must consider the implications of producing reusable information at each stage of the software development life cycle.

We begin by considering software requirements and domain analysis. We must determine the software elements that will make good candidates for future reuse during the system analysis stage. This process is referred to as *domain analysis.* An application domain analysis has some unique characteristics that are not normally addressed in a traditional analysis phase.[22] The goal of domain analysis is to increase the reusability of software components developed during the software engineering process. The key to domain analysis is thoroughly researching the domain to define the functions that the software must provide. Using these functions, we can develop a set of requirements and specific features. We can then abstract the related functions and features into reusable software components with well-defined interfaces.

Domain analysis follows good software engineering practices[4] and parallels the ideas of object-oriented design and analysis.[23] It is also possible to apply these principles to non-object-oriented software languages.

Domain analysis extends systems analysis and emphasizes:

- Identification of distinctive features of the domain
- Definition of relationships among domain elements
- Identification of those things that are common across applications within the domain
- Identification of unique features for each application

A domain analysis for reuse potential in an application must involve a perspective that looks beyond the development of a single project or system.[24] The domain analyst must examine the needs and requirements of a collection of systems or similar applications.[16] With this expanded form of analysis, reuse opportunities can be identified for a class of applications, rather than for a specific project.

T. Biggerstaff and C. Richter[13] discuss the domain of numerical computation routines as a classic example of successful software reuse. They state that the numerical computation domain is unique in that it (1) contains only a small number of data types, (2) is well understood, and (3) involves a static technology with slow, upwardly-compatible changes. The fact that the domain is largely static means that the reuse library can be quite stable, allowing repeated reuse over longer periods of time for different applications. A reuse domain where the underlying technology changes rapidly would not suggest a great opportunity for long-term reuse.

In addition to domain analysis, there are several possible design approaches when designing for reusability.

Object-based and object-oriented design are useful reuse design approaches. Designing with objects includes the construction of software systems as structured collections of abstract data-type implementations.[25] Objects encapsulate data and only operate on those data through well-defined operations or methods. Inheritance, generalization, and overriding behavior are key aspects of object-oriented modeling and design that enable objects to be easily reused and extended without modification to the original object. Several object-oriented modeling schemes have been defined and are in use today to describe and document the relationships among objects in a system. Modeling tools that use these schemes are becoming readily available. Such tools will facilitate reuse of object-oriented designs.

Generic or parameterized design can be used to effect design reuse. Generic software is designed to provide a generalized set of functions for use in different types of applications through the selection of parameters. However, the value of generalized designs is debatable. Building generality into reusable parts tends to be expensive and labor-intensive. The designer must predict and design in functions for use in future applications. Reusing a generalized design may also have performance, size, and complexity implications to the resulting system.

Message-oriented design is similar to object-oriented design in that software parts are interconnected through well-defined interfaces (messages).

The reuse of design information requires modeling tools and standard design templates and notations. Formalized specification methods can enhance the reusability of the design and provide precise functional definitions for use in the validation steps. Well-defined architectures and interfaces are critical to the reusability of a design.

During the implementation phase, the software developer can build reusable parts by developing structured code with well-defined interfaces and accurate documentation. The programmer should be careful to avoid environment-specific implementations. For example, code should be relocatable and conform to applicable standards. The reusability of the code is directly related to the reusability of the design and quality of other information for that code. As such, the reusable code parts and their documentation should be entered into the reuse library with links to the associated design and specification.

We can map the design, specification, and implementation methods for building reusable parts directly to the test phases. Reusable test designs for test cases and test tools, test specifications (often called test plans), and test code, including test cases, test scripts, and test documentation, can be produced for reuse in the same manner as reusable information for system products.

Software organizations investing in the production of reusable software information must also

acquire or invest in reuse library management tools. These tools assist in collecting and classifying the reusable software information for easy location and retrieval. The Reusable Software Library (RSL) prototype is an example of a design and programming tool for Ada.[26] This tool includes an underlying database, a library management system, a user query system, a software component retrieval and evaluation system, and a software computer-aided design system. The RSL classification scheme is based upon the assignment of hierarchical category codes combined with descriptive keywords.

Developing the business case for reuse

Whether an organization considers making investments in reusing software or producing reusable software, the decision to initiate a reuse program requires a convincing business case, based on realistic and quantifiable results. To develop a business case, the organization must gather, track, and statistically analyze software development information using metrics that meaningfully reflect the software development process. The organization combines these metrics with financial data, historical data, and the related costs and benefits of reuse to produce an expected return on investment upon which managers may base their business decisions.

Software metrics serve an important role in effective software management. However, the lack of an industry standard for reuse metrics results in one of the major inhibitors to a coordinated reuse program.[27] Without a means to quantify the practice, development organizations cannot judge their return on investment and therefore refrain from engaging in an active reuse program. However, if we use metrics in a return on investment model to verify and demonstrate the substantial benefits of reuse, we find organizations more receptive to a formal reuse program. With published productivity gains commonly claimed between 20 to 40 percent[28] and occasionally up to an order of magnitude,[29] organizations should want to take advantage of the increased output and corresponding lower costs that reuse offers.

Management traditionally uses metrics to assist in quantifying the software process. With an emerging technology, however, metrics must extend beyond their traditional role. Reuse metrics must also encourage the practice of reuse. We find most organizations do not practice formal reuse or resist investing in a formal reuse program. Reuse metrics can assist these organizations by providing favorable process improvement statistics and by placing emphasis on activity conducive to reuse. Our experiences show that we can successfully motivate managers by using a return-on-investment (ROI) model that shows value to their organizations.

Reuse metrics used in an ROI model can demonstrate substantial benefits.

Finally, reuse metrics must establish an effective standard that development organizations can implement. The organizations must be able to obtain the data easily and they must have the ability to implement and interpret the information in a meaningful, uniform way. In summary, reuse metrics must quantify reuse, encourage reuse, and standardize reuse counting methods.

To succeed in developing these goals and to contribute to a business case for reuse, we must look at the way the organization practices reuse. Factors influencing how organizations practice reuse include (1) how well the organization adheres to software development processes and (2) the management structure of the development group. These factors determine what the reuse metrics should record and the value of the activity.

How organizations practice reuse

The most fundamental difference in how organizations practice reuse lies in whether they are simply recovering old code for later use (i.e., unplanned reuse) and whether they engage in a formal, planned reuse program. These two classes are distinguished by when the organization makes the decision for reuse.[30] Planned reuse starts early in the software life cycle and involves a thorough requirements study and domain analy-

sis of the problem area. By doing this additional planning and domain analysis, organizations identify the factors that normally change in software. Examples are hardware or system software; user, mission, or installation; and function or performance.

Greater cost and productivity benefits result from planned reuse.

Early design and analysis results in components that can accommodate these changes without modification. However, this additional work requires time and effort. This investment in planned reuse results in an increased level of generality and quality in the initial development of the component. As subsequent projects reuse the generalized software, organizations can recover this investment quickly. Organizations also recover the investment through reduced support costs because they need to maintain only one reusable product rather than several nonreusable ones.

Unfortunately, traditional software development usually fails to plan for reuse. Although organizations may informally consider using existing software in a new application, they develop most new software from scratch. Independently of how often organizations informally use previously developed software in new applications, traditional software development methods do not include the systematic reuse of existing code. To determine the financial return of a reuse program, we must distinguish between these two classes of reuse.

Code recovery. We call copying and modifying existing code to meet new requirements *code recovery*. Because code recovery proliferates new software and results in additional products to maintain, it has nominal benefits as compared to planned reuse. Nonetheless, many organizations practice several forms of code recovery to meet their development objectives and schedules. The following code recovery processes show how organizations accommodate the three change factors of hardware or system software; user, mission, or installation; and function or performance.

Rehosting occurs when organizations modify existing software to fit new hardware or system software. Rehosting focuses on revising internal interfaces to fit the new environment, thereby effecting minimal change in function.

Retargeting occurs when organizations modify existing software to fit a specific use or installation. Retargeting focuses on modifying external interfaces and physical configurations of equipment. The code function does not change, although implementation details in the code do change.

Salvaging occurs when organizations extract potentially useful software from an existing system and modify it to fit a new use. Salvaging is the most basic form of recovery; it relies on a bottom-up strategy of integrating elements from many sources to build a new product.

In each of the above situations, organizations copy and modify the original software to create new software. This adds to the maintenance and development costs for the product. However, organizations can still benefit from recovering old code, especially if the cost of custom development greatly exceeds the cost of modification. Unfortunately, every organization that recovers the code must incur the cost of modification and must maintain its copy of the software.

Planned reuse. If organizations plan for reuse, they can make software components that they can readily adapt to the three change factors listed above. For example, if they exclude system dependencies and use parameters to control environment variables, the component may find use in another product. Planned reuse increases the value of software by expanding its applicability. This requires domain analysis, careful design, and building into the software tailorable attributes based on a range of potential uses. The following show ways organizations plan for reuse and accommodate the three major change factors.

Porting occurs when an organization moves a software item from one hardware or software system to another. Ease of porting results from

design considerations that isolate machine-dependent functions and use standard virtual interfaces.

Tailoring allows a single software system to adapt to the needs of specific installations, users, or missions. Tailoring occurs when an organization plans product modifications through a controlled customization interface that does not involve direct source code changes. A system designed for tailoring typically uses a generic or parameterized design approach, generic modules, or changes controlled through inheritance.

Assembling occurs when an organization constructs a software system with prebuilt parts. Many refer to this form of reuse as the composition or building-block approach. Assembly is the most common form of formal reuse. Using this strategy, organizations design, code, test, and document software components for integration.

Table 1 provides a summary of how system changes relate to the classes of reuse.

Measuring code recovery and planned reuse. Because planned reuse results in fewer products to maintain and avoids modification costs, planned reuse provides greater cost and productivity benefits than code recovery. The benefits accrue rapidly over several development cycles as more organizations reuse the software. Using these criteria, we believe metrics should focus on planned reuse. This does not mean that code recovery does not serve a useful and important role in software development. However, the benefits only come during the development phase and quantifying these benefits can become very difficult and subjective. For example, what value do we place on modifying a small portion of components versus a large portion? How do we collect data on the portion of components modified? How do we estimate the resulting development savings? Although others have studied these issues,[31] we do not have these data nor can we justify investing in tools and process overhead to collect them. Because we believe code recovery has limited benefits when compared with planned reuse and because our goals include adopting formal reuse in our software process, we do not factor the three techniques for code recovery into our reuse metrics. However, some organizations, such as the IBM Federal Systems Company, track the amount of recovered code in their products to emphasize the amount of "total leverage" they gain by copying and modifying old software.

Table 1 Relation between system changes and classes of reuse

Required Change	Code Recovery Only	With Planned Reuse
Hardware or system software	Rehosting	Porting
User, mission, or installation	Retargeting	Tailoring
Function or performance	Salvaging	Assembling

Of the three forms of planned reuse, we find many organizations assembling reusable components into new applications. Language features such as generics, parameterization, message passing, and inheritance control all allowable modifications to component function. In fact, this is the current state of reuse technology and reuse metrics must capture code assembly. The next most advanced form of reuse comes with tailoring, which we find most often in organizations with established reuse programs. These organizations conduct domain analysis and carefully design programs for reuse. Reuse metrics must also capture this activity. An example of tailoring comes from the tremendously successful reuse experiences on the IBM Advanced Automation System (AAS) for the Federal Aviation Administration.[32]

The third form of planned reuse comes from porting. However, porting causes difficulty when determining the investment value of reuse because porting is already factored into the business planning of products. Planners normally estimate resources to develop a product on one hardware platform or operating system and then allocate a relatively nominal amount of resources for changes required to adapt to other environments.

Since porting normally involves adapting a minor portion of a large product, to include ported code in reuse metrics would cause misleading results in the form of unrealistically high measures of reuse activity. For example, an organization making small changes to a large base might report levels of reuse close to 100 percent, whereas an organization performing an equal amount of labor on an original project might do very well to demonstrate reuse levels of 5 to 10 percent. Furthermore, porting an application program to a new operating system by a simple recompilation of the

Table 2 Reuse techniques and reuse metrics

Code Recovery	Measured	Planned Reuse	Measured
Rehosting	No	Porting	Separately
Retargeting	No	Tailoring	Yes
Salvaging	No	Assembling	Yes

source code has a different value to the organization than does avoiding having to write a similar amount of code for a custom application. To prevent distortions caused by porting large programs and determining their investment value, we do not include porting in these reuse metrics. This allows us to isolate and quantify the benefits of reuse beyond those of established business and software engineering practice. Some organizations that port large amounts of software separately track and report the amount of porting on their products. Table 2 shows how we measure the various classes of code recovery and planned reuse.

When to measure reuse. We discussed how to distinguish between code recovery, which results in new software to maintain, and planned reuse, in which organizations assemble or tailor products from building blocks of reusable software. Next, we define reuse based on who uses the component.

Experience shows us that we can expect good program design and management within development organizations. However, coordination and cooperation between organizations, especially as their size increases, becomes less likely. Communication, necessary for the simple exchange of information and critical to sharing software, becomes more difficult as the number of people involved grows and natural organizational boundaries emerge. We find that to improve the practice of reuse we must develop metrics that encourage reuse across these organizational boundaries.

We define a reused component as one used by an organization that did not develop or maintain the component. We expect an organization to use the code it develops. Because we seek to quantify the financial benefit accrued by effort saved, we place a value on avoiding program development by using another organization's work. Because software development organizations can vary, we define a typical organization as either a programming team, department, or functional group of about eight people. Also, although organizational size can indicate how well communication within and between organizations takes place, we find functional boundaries equally important. For example, a small programming team may qualify as an organization if it works independently, so our development organizations typically range from 4 to 20 persons.

For consistency, we consider the type and size of the reporting organization as part of the metrics. This provides us with an informal check on the flexibility allowed in selecting the most appropriate boundary for the organization. Selection of an inappropriately small boundary would distort the value of the metrics upward and an inappropriately large boundary would result in low reuse values. Changing the organizational boundary between reports would eliminate any possibility for comparisons and evaluation of the reuse program.

Reusing versus using components. Most organizations report their reuse effort as the reuse percent of a product. The reuse percent comes from the portion of the product (normally expressed in lines of code, or LOC) that the organization avoided having to write by reusing software. The effort attributed to reuse comes from completely unmodified reusable components. We can easily identify reusable components in new products because we use straightforward criteria. If use of a component saves having to develop a similar component, we record it as reuse. However, although an organization may "use" a component numerous times, it can reuse a component but once.

Accurate estimates of the benefits of reuse and return on investment analysis of projects depend on this distinction. Because we expect organizations to use components previously developed for a product or previously developed by themselves, we do not credit the organizations with reuse savings that result. In other words, source instructions from a reused part count once per organization, independently of how many times one calls or expands the part. There are two reasons for this: (1) Metrics must accurately reflect effort saved. Programmers use subroutines and macros because many functions are repetitive. (2) Metrics should not depend on the implementation.

The choice of using a subroutine versus a macro should result from design considerations well outside the realm of reuse. A programmer should not decide to use macros because multiple in-line expansions increase the amount of reuse reported on a project. In other words, the decision to reuse should make good business sense.

In one actual example, a project reported 11 thousand lines of code of reuse on a relatively small application. Closer inspection revealed 5120 lines of the 11 thousand lines of code came from one 10-line reusable macro and that all 5120 lines came from the same module. A code review revealed that the original code:

```
Do i := 1 to 512
        MACRO(i);
```

consisted of two instructions (the DO . . . WHILE and the call to MACRO) and the 10 reused instructions from MACRO. However, to optimize the loop, it was unrolled to yield:

```
        MACRO(1);
        MACRO(2);
          ....
        MACRO(511);
        MACRO(512);
```

The reuse report therefore contained 512 source instructions and 5120 reused instructions, which does not accurately reflect the productivity or reuse on the project.

Units of measurement. We express these metrics using traditional lines of code to quantify the effort in software development. Although lines of code have well-known deficiencies as a unit of measure,[33,34] their universal use makes them simple to understand, easy to collect and compare, and difficult to distort. Nonetheless, we take several actions to increase our confidence in lines of code as a unit of measurement. These actions include use of a standard code counting tool. Another action eliminates the units of LOC from the metrics by using metrics derived from ratios and percentages of effort. For example, reporting levels of reuse as a percent of the delivered product or reporting programming leverage using a productivity ratio reduces concerns about the underlying unit of measure. This approach also allows organizations that use other units of measurement, such as function points, to express their reuse activity and results without any changes to the metrics.

A historical perspective on reuse business cases

Several groups have conducted research in the area of reuse metrics and business-case models. In 1988 Gaffney and Durek[35] published a comprehensive model addressing business-case analysis of reuse. They premise their model on the need to amortize the cost of the reuse program, including the additional cost to build reusable components, across all projects using the component. When doing cost-benefit analysis for software reuse, one needs to consider the long-term benefits and associated costs, which apply to every project using the component. Taking a short-term approach to these costs, the additional cost of developing reusable components greatly overemphasizes their cost relative to their benefit. The authors argue that a better economic estimate includes the number of times the component is reused.

Gaffney and Durek define the cost of software development with reuse relative to the cost of software development with all new code.[36] They developed an equation for *relative cost, C*:

$$C = R_U \times 1 + R \times \left(b + \frac{E}{n}\right)$$

where:

R_U is the portion of nonreused (newly written) code.
R is the portion of reused code.
b is the relative cost of integrating reused code.
E is the relative cost of creating reusable code.
n is the number of uses over which the reused code is to be amortized.

If the R value is zero (e.g., there is no reuse), the value of C is equal to 1. The equation also shows that for a reusable component to pay off, the component must be reused at least two times.

The second metric defined in Reference 35 is the *productivity index, PI*, which is the productivity relative to that of creating the software product without reuse. The productivity index is defined as the inverse of C:

$$\textit{Productivity index} = \frac{1}{C}$$

A *PI* of 2.5 indicates the measured project was 150 percent more productive in terms of cost than the project would have been without reuse.

Observing that the coefficient b varies depending on the type of reuse (recovering, porting), Reference 32 extends the reuse value added metric model by defining additional values of b for the different types of reuse, R. C becomes $R_U \times 1 \times R_i \times b_i$ for each (R_i,b_i).

For example:

R_0 is the portion of reused code from other sources.
b_0 is the relative cost of integrating reused code from other sources.
R_R is the portion of code requiring re-engineering (copied and modified code).
b_R is the relative cost of integrating re-engineered code.

An additional cost-benefit model of reuse is presented as the NATO model in References 37 and 38. The NATO model consists of listing the major benefits and costs of reuse and then applying time-value of money formulas to adjust for future values. The benefits are:

- Saving due to avoided cost, S_R—the sum of costs avoided each time the component is reused
- Service life, L—the useful lifetime, in years, of the component
- Demand—the number of times the component is likely to be reused during its service life, L

The costs of reuse are:

- Cost to reuse, C_R—the cost incurred each time the component is reused, including identification, retrieval, familiarization, modification, and installation (this is the relative cost of reuse)
- Accession time, T_A—the amount of time likely to elapse between the decision to acquire the component and its availability in the library
- Accession cost, C_A—the cost to add the component to the library, including obtaining raw material developing the complete component, and installing it in the library
- Maintenance cost, C_M—the cost to maintain the component in the library, including maintenance and change distribution

The net saving to the reuser (NSR) is the difference between the savings due to avoided cost and the cost to reuse:

$$\text{NSR} = S_R - C_R$$

The net saving to the supported program (NSP) is the total savings from all instances of reuse of a component less the accession and maintenance costs. The total savings from all instances of reuse is the NSR multiplied by the number of reuses, N:

$$\text{NSP} = (\text{NSR} \times N) - (C_A + C_M)$$

Although the NATO model continues with adjustments for the time value of money, there is little guidance on collecting the data required for the model. For example, there are no details on accounting for the savings due to avoided cost, how to estimate the number of times a component is likely to be reused, nor estimating the service life of a product that does not wear out. Finally, where data are not available or the analyst feels mitigating factors affecting the risk of the reusable product exist, statistical distributions and estimates of risk factors may be used to adjust the inputs.

In 1992 Gaffney and Cruickshank[39] published a generalized model which differentiates between costs of domain analysis and application engineering. The basic reuse cost equation is defined as

$$C_S = C_{US}S_S = \frac{C_{DE}S_T}{N} + C_{VN}S_N + C_{VR}S_R$$

where:

C_S is the total cost of the application system.
C_{US} is the unit cost of the application system.
C_{DE} is the unit cost of domain engineering.
S_T is the expected value of the size of the reuse library measured in source statements.
N is the number of application systems over which to amortize the domain engineering costs.

C_{VN} is the unit cost of new code developed for the application system.
S_N is the number of new source statements written for the application system.
C_{VR} is the unit cost of reusing code from the reuse library for the application system.
S_R is the number of reused source statements incorporated into the application system.
S_S is the total number of application source statements.

The researchers also define a library efficiency metric $E = S_R/S_T$, which measures the degree to which maximum use has been made of the reuse library. If full use has been made of the library, then $E = 1$ and the basic reuse cost equation reduces to:

$$C_{US} = C_{VN} - \left[C_{VN} - C_{VR} - \frac{C_{DE}}{N}\right] R$$

where R is the proportion of code that is reused code. The authors go on to apply the model to example applications and demonstrate significant increases in overall productivity as a result of increases in new code productivity and the percent of reuse.

Reuse metrics

Ultimately, the goals of the organization define what we measure and report as reuse.[40] Because the IBM metrics are inputs to an ROI model, we need metrics that reflect effort saved, both by quantifying the level of reuse in an organization and by determining the investment value of reuse. We also want to encourage reuse activity beyond the good software engineering practices already established in the company. Specifically, the IBM reuse metrics are intended to reflect the effort saved and encourage reuse.

Of the reuse metrics developed by others, few provide definitions and supporting information on how to gather the data required to implement their metrics. Reference 29 differentiates between reuse within an organization and reuse from sources external to the organization. No other paper addresses ways to measure the classes of reuse or provides a concentrated definition of RSI. We also have other considerations, such as the availability or ease of collecting the required data. Without this necessary detail we cannot responsibly determine the accuracy of existing reuse metrics and their related ROI models.

Because we wanted to address these considerations, we started by collecting the data we had and by defining the data we needed. We then built an ROI model using those data. By carefully defining what we count as reuse and the value we attach to it, we have a high degree of confidence in how the metrics motivate reuse and the accuracy they give to our business model.

Metrics are used to support practices, processes, and goals.

The following sections define the reuse metrics used in IBM.[41] We developed these metrics and an ROI model to support our business practices, software development process, and reuse goals.[42] However, these metrics also support companies with similar business practices and reuse goals.

Observable data. We calculate the reuse metrics presented in the next section from the following observable data elements. Although most of these data elements have been collected by IBM for many years, they are similar to the data elements collected by other companies.[43] We can usually measure directly observable data from the product. For example, we routinely count the different classes of source instructions. Observable data may also come from historical information. For a variety of reasons related to managing the software development process, IBM has also collected information on such things as costs for software development and statistical error rates. Detailed descriptions of each of the following required observable data elements are given after the summary in Table 3:

- Shipped source instructions (SSI)—total lines of code in the product source files
- New and changed source instructions (CSI)—

Table 3 Observable data

Data Element	Symbol	Unit of Measure	Source
Shipped source instructions	SSI	LOC	Direct measurement
Changed source instructions	CSI	LOC	Direct measurement
Reused source instructions	RSI	LOC	Direct measurement
Source instructions reused by others	SIRBO	LOC	Direct measurement
Software development cost	Cost per LOC	$/LOC	Historical data
Software development error rate	Error rate	Errors/ LOC	Historical data
Software error repair cost	Cost per error	$/Error	Historical data

total lines of code new or changed in a new release of a product

- Reused source instructions (RSI)—total lines not written but included in the source files. RSI includes only completely unmodified reused software components.
- Source instructions reused by others (SIRBO)—total lines of code that other products reuse from a product
- Software development cost—a historical average required for estimating reuse cost avoidance
- Software development error rate—a historical average required for estimating maintenance cost avoidance
- Software error repair cost—a historical average required for estimating maintenance cost avoidance

Shipped source instructions. Shipped source instructions (SSI) come from the number of noncomment instructions in the source files of the first release of a product. SSI does not include reused source instructions (RSI). A call to a reusable part counts as one SSI. When reporting reuse measures for development organizations, SSI includes all the source instructions the organization maintains.

Changed source instructions. Changed source instructions (CSI) come from the number of noncomment source instructions that an organization adds, modifies, or deletes in a subsequent release of a product. CSI does not include reused source instructions (RSI) or unchanged base instructions from prior releases of the product. CSI includes source instructions from partially modified components incorporated into the release. A call to a reusable part counts as one CSI.

Reused source instructions. Reused source instructions (RSI) come from source instructions shipped, but not developed or maintained by the reporting organization. RSI serves as our primary observable measure of reuse consumption. RSI comes from completely unmodified components. Base instructions from prior releases of a product do not count as RSI; we do not consider the second release of a product as having reused all code from the first release.

In a hierarchical reporting structure, we obtain the RSI for higher levels of management by summing the RSI values reported by their suborganizations. We also use this technique to calculate RSI when several organizations work on a single product. Because we previously defined organizations in such a way that we do not normally expect them to share software, it is possible for a component to count more than once in the RSI of the higher level organization; this indicates that more than one suborganization reused the same component. Because we also roll up the values for SSI and CSI in this manner, the higher level manager receives a weighted average of the levels of reuse in each of the suborganizations.

Source instructions reused by others. Source instructions reused by others (SIRBO) for an organization come from source instructions reused by other organizations. SIRBO serves as our primary observable indicator of reuse production; it reflects how much an organization contributes to reuse. For a reuse program to succeed, organizations must not only reuse software but help other organizations reuse software. SIRBO not only measures the parts contributed for use by others but also the success of those parts. Organizations writing successful reusable parts will have a very high SIRBO, because SIRBO increases every time another organization reuses their software. This encourages organizations to generate

high-quality, well-documented, and widely applicable reusable components.

We calculate SIRBO by summing over all parts that an organization contributes for reuse as follows:

SIRBO = (source instructions per part)
× (number of organizations using part)

As an example, an organization's contributions to a reuse library are: a 5 thousand-lines-of-code module in use by five other departments, a 15 thousand-lines-of-code macro in use by six other departments, and an unused 50 thousand-lines-of-code macro. The organization's SIRBO is expressed as follows:

SIRBO = (5 departments × 5 KLOC)
+ (6 departments × 15 KLOC)
+ (0 departments × 50 KLOC)
= 115 KLOC

where KLOC = thousand lines of code.

For the same reasons used in determining RSI, we calculate SIRBO independently of the number of times the same organization invokes or calls the part. The same rules apply that apply for counting RSI: use of a reusable part saved having to develop the part one time, not one time for every call to the part. SIRBO grows over time. As more organizations reuse the components, the SIRBO of the donating organizations increases. We base our SIRBO measurement on the most current list of reusers by using a process similar to software licenses.

Software development cost. To determine the financial benefit of reuse, we must know the cost of developing software without reuse. The new software development cost (cost per LOC) comes from historical averages that we normally obtain from the financial planners and management of the organization. If necessary, we calculate the new software cost by adding all the expenses of the organization, including overhead, and dividing by the total output (in LOC) of the organization.

Software development error rate. No amount of testing, inspection, or verification can guarantee the release of a product without errors (error rate). Although emphasis on quality and strict adherence to development processes leads to better products, errors inevitably reveal themselves after product release to the marketplace. Every development organization has a historical average number of errors uncovered in its products.

Note that we usually design and test software components built for reuse to stricter standards than those for normal program product components. We justify the additional cost of testing and quality assurance for reuse by the savings gained when other organizations do not have to develop and maintain a similar component. The additional testing not only helps identify errors in the component, but as more organizations reuse the component the more confidence we have in its quality.

Observable data elements are combined to form derived metrics.

Software error repair cost. To quantify the benefit of the increased quality of reusable components, we need the historical average cost of maintaining components with traditional development methods (cost per error). As with software development cost, we generally obtain this figure from financial planners and management in the organization. If necessary, we calculate the software error repair cost by taking the sum of all costs of repairing latent errors in software maintained by the organization, including overhead, and dividing by the number of errors repaired.

Derived metrics. The observable data elements combine to form three primary derived reuse metrics: reuse percent, reuse cost avoidance, and reuse value added. We also define a fourth metric, additional development cost, to complete our ROI model. As shown below, the first two metrics indicate the level of reuse activity as a portion of

Table 4 Derived metrics

Metric	Symbol	Derived from	Unit of Measure
Reuse percent • For products • For product releases • For organizations	Reuse percent	 • SSI, RSI • CSI, RSI • SSI, RSI	Percent
Reuse cost avoidance	RCA	SSI or CSI, RSI, cost/LOC, error/LOC, cost/error	Dollars
Reuse value added	RVA	SSI, RSI, SIRBO	Ratio
Additional development cost	ADC	Code written for reuse by others, cost/LOC	Dollars

effort and by financial benefit. The third metric includes recognition for writing reusable code. The fourth ROI metric accounts for added expenses directly attributable to producing reusable software. Table 4 provides a summary of these metrics.[41]

- Reuse percent—the primary indicator of the amount of reuse in a product or practiced in an organization. Reuse percent is derived from SSI, CSI, and RSI.
- Reuse cost avoidance—indicator of reduced total product costs as a result of reuse in the product. Reuse cost avoidance is derived from SSI, CSI, RSI, error rates, software development cost (cost per LOC), and maintenance costs (cost per error).
- Reuse value added—an indicator of leverage provided by practicing reuse and contributing to the reuse practiced by others. Reuse value added is derived from SSI, RSI, and SIRBO.
- Additional development cost—indicator of increased total product costs as a result of developing some product code for subsequent reuse by others. Additional development cost is derived from the amount of code written for reuse by others and normal software development cost (cost per LOC).

Reuse percent. The purpose of the reuse percent measurement is to indicate the portion of a product, product release, or organizational effort that can be attributed to reuse. The ease of calculating and understanding reuse percent makes it an important metric. Unfortunately, many companies report their reuse experiences in terms of reuse percent but few describe how they calculate the values. They commonly include informal reuse in the metric, making it difficult to assess actual savings or productivity gains. Inasmuch as we provide a supporting framework and clearly define what we mean by RSI, we believe the reuse percent metric reasonably reflects real effort saved.

Reuse percent of a product. Now that we have defined how we obtain our data, we use a simple percent equation to calculate the reuse percent of a product (or first release of a product):

$$\text{Reuse percent} = \frac{\text{RSI}}{\text{RSI} + \text{SSI}} \times 100 \text{ percent}$$

Consider the following example. If a product consists of 75 KLOC SSI and an additional 25 KLOC from a reuse library, then the reuse percent of the product equals:

$$\text{Reuse percent} = \frac{25 \text{ KLOC}}{25 \text{ KLOC} + 75 \text{ KLOC}} \times 100 \text{ percent} = 25 \text{ percent}$$

Reuse percent of a product release. For a new release of a product, we calculate the reuse level based on work done since the last release of the product. We exclude all code in the previous releases of the products (the *product base*) and count RSI from reusable components added to the product for this release. A call to a component used in a previous release is a new or changed source instruction (CSI). We still use a simple percent equation but substitute CSI for SSI:

$$\text{Reuse percent} = \frac{\text{RSI}}{\text{RSI} + \text{CSI}} \times 100 \text{ percent}$$

The following is a product release example. If a new release of a product consists of 8K CSI plus 2K new RSI from a reuse library, the reuse percent for this product release equals:

$$\text{Reuse percent} = \frac{2 \text{ KLOC}}{2 \text{ KLOC} + 8 \text{ KLOC}} \times 100 \text{ percent} = 20 \text{ percent}$$

Reuse percent for an organization. Often we would like to know how an organization as a whole practices reuse, without having to consider the number of products or parts of products it develops. For an organization, all software developed and maintained by the organization counts as the SSI of the organization. Any software used by the organization but maintained elsewhere counts as RSI. The reuse percent equation remains the same:

$$\text{Reuse percent} = \frac{\text{RSI}}{\text{RSI} + \text{SSI}} \times 100 \text{ percent}$$

Here is an organizational example of reuse percent. If a programming team develops and maintains 70K SSI and the team additionally uses 30K RSI from a reuse library, the reuse percent for the team equals:

$$\text{Reuse percent} = \frac{30 \text{ KLOC}}{30 \text{ KLOC} + 70 \text{ KLOC}} \times 100 \text{ percent} = 30 \text{ percent}$$

Reuse cost avoidance. The purpose of a reuse cost avoidance (RCA) measurement is to quantify the financial benefit of *reusing software.* The ability to show the return on investment potential of reuse makes this a particularly important metric. Although we also use RCA in parts of our corporate ROI analysis for reuse, we find RCA helps with the insertion of reuse at all organizational levels.

The potential benefits and savings of reusing software depend on the specific project and reuse percent. Even organizations that only consume reusable software must make investments in process changes, tools, and education. For example, for software developers and managers who require training in software reuse, the amount of training may require a substantial commitment. Reference 44 discusses an experiment to test whether programmers untrained in software reuse can accurately assess component reusability. The experiment concludes that software development personnel untrained in software reuse cannot assess the worth of reusing a candidate component. Further, the subjects of the experiment were influenced by unimportant features and were not influenced by important features of reusable software.

One method for determining the RCA for a consumer of reusable software involves identifying and quantifying the individual costs and benefits associated with the incorporation of the reused software into the system. With this itemized approach to computing RCA, we can calculate the total reuse cost avoidance as follows:

$$\text{Reuse cost avoidance} = \sum_{i=1}^{j} b_i - \sum_{i=1}^{k} c_i$$

where b_i is a benefit and c_i is a cost associated with being a consumer of the reusable software.

Itemized costs and benefits vary for each project and organization. Figure 1 lists examples of benefits and costs associated with reusing software information. The benefits of reusing software typically far outweigh the costs and include the dollars saved by not having to design, develop, document, test, maintain, and manage the development of the reused software. Benefits may also include reduced cost of tools or equipment that would have otherwise been required, if the software had been developed rather than reused. The benefits may take the form of additional revenue from delivering a product to market earlier or improved customer satisfaction.

Costs of reuse include such things as dollars spent to educate the organization on software reuse and the availability of reusable software information. The organization may also incur costs for a system and application domain analysis and time spent identifying portions of the design to make into candidates for reused software. Programmers require time and facilities to select reused software. If the organization obtains commercially available software packages, it may have to pay license fees or purchase the software. Also if an organization decides it must modify or customize the software, the typical development

Figure 1 Benefits and costs of reusing existing software information

REUSE CONSUMER BENEFITS:	MEASUREMENT
b_1 Reduced cost to design	person months × $/person month
b_2 Reduced cost to document (internal)	pages × $/page
b_3 Reduced cost to implement	person months × $/person month
b_4 Reduced cost to unit test	person months × $/person month
b_5 Reduced cost to design tests	person months × $/person month
b_6 Reduced cost to document tests	pages × $/page
b_7 Reduced cost to implement test cases	person months × $/person month
b_8 Reduced cost to execute testing	person months × $/person month
b_9 Reduced cost to produce publications	pages × $/page
b_{10} Added revenue due to delivering product sooner to the market place	months × $/month
b_{11} Reduced maintenance costs	errors × $/error
b_{12} Added revenue due to improved customer satisfaction with product quality	sales × $/sale
b_{13} Reduced cost of tools	$
b_{14} Reduced cost of equipment	$
b_{15} Reduced cost to manage development and test	person months × $/person month
REUSE CONSUMER COSTS:	
c_1 Cost of performing cost-benefit analysis	person months × $/person month
c_2 Cost of performing domain analysis	person months × $/person month
c_3 Cost of locating and assessing reusable parts	person months × $/person month
c_4 Cost of integrating reusable parts	person months × $/person month
c_5 Cost of modifying reusable parts	person months × $/person month
c_6 Cost of maintaining modified reusable parts	errors × $/error
c_7 Cost of testing modified reusable parts	person months × $/person month
c_8 Fees for obtaining reusable parts	$
c_9 Fees or royalties for reusing parts	copies used × $/copy
c_{10} Cost of training on software reuse	$

costs associated with making it ready to use add to the costs, and many of the benefits become lost.

If we can identify and quantify each of the individual costs and benefits, we can accurately derive the RCA from the itemized cost-benefit list. However, we often do have enough information or cannot justify the expense of obtaining it. Organizations new to software reuse may have difficulty estimating the costs of integrating the reused parts into the system because they do not have historical data upon which to base the estimates. We ourselves find it difficult to quantify intangible items such as improved revenue from earlier delivery of the software or customer satisfaction.

To determine RCA we have found a useful method that depends on an estimate of the effort required to integrate reused software. Experience and studies show that we can estimate the cost of this effort at only about 20 percent of the cost of new development.[45,46] This percent assumes no modification of reusable parts. If the organization must modify or maintain the modified reusable parts we adjust this percent upward. Based on this relative cost of reuse, we define the financial benefit attributable to reuse during the development phase of a project as 80 percent of the cost of developing new code. We call this benefit the development cost avoidance (DCA):

Development cost avoidance

$$= \text{RSI} \times (1 - 0.2) \times (\text{new code cost})$$

$$= \text{RSI} \times 0.8 \times (\text{new code cost})$$

Development, however, comprises only about 40 percent of the software life cycle.[47] There is also a significant maintenance benefit that results from reusing quality software. We can quantify this benefit as the cost avoidance of not fixing errors in newly developed code,[35] and we define this benefit as the service cost avoidance (SCA):

Service cost avoidance

= RSI × (error rate) × (cost per error)

The total reuse cost avoidance is expressed as follows:

Reuse cost avoidance (RCA)

= development cost avoidance (DCA)

+ service cost avoidance (SCA)

As an example of this concept, if an organization has a historical new code development cost of $125 per line, an error rate of 1.25/KLOC, and a cost to fix an error of $20K, then the estimated RCA for integrating 20K RSI into a product equals:

Reuse cost avoidance

= (20 KLOC × 0.8 × $125 per line)

+ (20 KLOC × 1.25 error per KLOC

× $20K per error)

= $2.0 million + $0.5 million

= $2.5 million

Reuse value added. The previous two metrics measure how much organizations reuse software. We must also motivate the producer side of software reuse by recognizing contributions to the inventory of reusable software. The reuse value added (RVA) metric provides a way to recognize organizations that both reuse software and help other organizations by developing reusable code.

Someone must produce the software for everyone to reuse. Some development groups recognize this and organize to obtain the most benefit possible from both consuming and producing reusable software. For example, the IBM Mid-Hudson Valley Programming Laboratory and the IBM Federal Systems Company in Rockville, Maryland, dedicate programming teams to develop and maintain shared software or site-wide reuse libraries. Corporate parts centers, such as the Böblingen software center, also develop and maintain software for IBM-wide use. Experience shows that although these types of groups may have modest values for the reuse percent metric, these groups have extremely high values for the RVA metric, and this high RVA indicates the tremendous programming leverage they provide to their organizations.

We use a ratio, or productivity index, to represent the RVA. Organizations with no involvement in reuse have an RVA = 1; an RVA = 2 indicates that the organization has doubled its effectiveness through reuse. That organization has become twice as effective to the corporation because it either directly or indirectly produced more software than it could without reuse. Therefore, the total effectiveness of a development group is:

$$\text{Reuse value added} = \frac{(\text{SSI} + \text{RSI}) + \text{SIRBO}}{\text{SSI}}$$

Consider the programming team that maintains 80 KLOC and uses 30 KLOC from a reuse library. If five other departments reuse a 10 KLOC module the programming team contributed to the organizational reuse library, the RVA of the programming team is:

Reuse value added

$$= \frac{(80\ \text{KLOC} + 30\ \text{KLOC}) + (5\ \text{dept.s} \times 10\ \text{KLOC})}{80\ \text{KLOC}}$$

= 2.0

In this example, the RVA of 2.0 indicates the programming team became 2.0 times more effective as a result of reuse.

Additional development cost. Developing software intended for reuse costs the reuse producer more than developing code for one-time use only. The organization must spend additional effort to ensure that the code is made ready for reuse in different application domains. The additional development cost (ADC) metric seeks to quantify this effort.

As with RCA, we can determine the ADC by identifying and quantifying the individual costs and benefits associated with producing reusable software. With this itemized approach to computing ADC, the total is expressed as follows:

$$\text{Additional development cost} = \sum_{i=1}^{j} c_i - \sum_{i=1}^{k} b_i$$

where c_i is a cost and b_i is a benefit associated with being a producer of the reusable software.

The degree of investment in building software for reuse varies, depending on the needs and priorities of each software organization. Through ongoing education and incentives, the software manager can promote the production of reusable software within the organization. The costs to the organization include several factors:

- Domain analysis required to conduct a thorough study of the problem and reveal opportunities for reuse. To be practical, the domain analyst must have an in-depth knowledge of the application domain and training or experience in software reuse and design.
- Training required for software developers to learn the concepts and practices of building reusable software. Software reuse training includes learning concepts of data encapsulation, information hiding, constructing well-defined interfaces, using language-specific features, and programming for environment-independence. These important software reuse attributes have analogies in object-oriented design and analysis.
- Library tools and maintenance required for a library to store reusable components. This library may require additional hardware and software tools for library access and parts retrieval.
- Development and certification of reusable software required to ensure that the parts are designed, implemented, and tested for reuse in other environments. The organization must also place extra emphasis on user documentation so the reuser can understand the function of the software and its interfaces. For high-quality, reusable components, we require certification by an independent test group to ensure software quality and function.
- Involvement and communication with other software development groups (both internal or external) to help locate potential sources of reusable software information. Each organization investing in software reuse should assign personnel the responsibility for staying abreast in software reuse developments and new technology. This requires time to read and investigate literature, participating in work groups or seminars, and communicating information back to the software organization.
- Encouraging participation to avoid many of the inhibitors to widespread software reuse. One source of inhibitors are cultural or social issues.[6] Managers must provide incentives for participating in software reuse to break through some of these barriers. In design activities, the manager should encourage design for reuse by rewarding designers for the reuse of their designs in multiple applications. In addition to measuring the typical lines of code produced, the manager can measure and reward lines of code built for reuse.

Fortunately, the additional costs of producing reusable software include some direct benefits. An organization may realize these benefits through cost recovery or collection of fees and royalties from reusers of the software they produce. Figure 2 lists examples of benefits and costs associated with producing reusable software information.

We call the sum of the costs and benefits of producing reusable information the relative cost of writing for reuse. We set the relative cost of writing for reuse with respect to the cost of writing code for one-time use, which we take equal to 1. As with RCA, we often find the costs and benefits of producing reusable software difficult to identify and quantify. We may also find we do not have all the required information or cannot justify obtaining it. To estimate the ADC, we use our experience to show we can estimate the cost of this additional effort at about 50 percent of the cost of new development.[45,46] Therefore, we define the additional development cost as:

$$\text{ADC} = (\text{relative cost of writing for reuse} - 1) \times \text{code written for reuse by others} \times \text{new code cost}$$

To illustrate this, take a programming team that develops and maintains 80 KLOC, of which 20 KLOC consists of macros and modules that the programming team contributed to the organizational reuse library. If the programming team has a historical new code development cost of $125 per line, and the relative cost of writing for reuse is 1.5, the ADC for the programming team is:

$$\text{ADC} = (1.5 - 1) \times 20 \text{ KLOC} \times \$125 \text{ per line} = \$1.25 \text{ million}$$

In this example, the ADC of $1.25 million reflects the investment made by the programming team to develop software that other programming teams can later reuse.

Reuse return on investment

The reuse metrics aid in both quantifying and standardizing counting methods for projects and development organizations. Because financial results are always a high priority, however, the most effective way to encourage reuse is to show the return on investment for reuse. This section describes a traditional way to evaluate return on investment through cost-benefit analysis. We combine the principles of cost-benefit analysis with the metrics described in this paper to provide templates for project- and corporate-level reuse business cases.

Calculating ROI using cost-benefit analysis. Cost-benefit analysis is a technique that uses estimates to compare and weigh the costs and benefits of an undertaking.[48] We can use cost-benefit analysis for software reuse investment decisions in three ways. As a planning tool, cost-benefit analysis assists in determining the appropriate amount of resources to apply toward software reuse investment. This analysis serves as an auditing tool for evaluating existing projects that practice software reuse. It also provides quantitative support to influence decisions on software reuse investments and strategies.

The most difficult task in a cost-benefit analysis is to assign values to the costs and benefits. Where possible, we must quantify intangible items. For example, improved quality can result in improved customer satisfaction, which in turn can result in revenue from additional sales. We often cannot predict these kinds of effects. When uncertainty surrounds the value applied of an intangible item, we can apply a range of values to that item and perform a sensitivity analysis. In a sensitivity analysis, we complete a cost-benefit analysis at the high and low ends of the range to determine the effect that the item has on the overall results. If we do not see a significant difference in the outcome over the range of values, we estimate the value within that range.

The following steps apply when completing a cost-benefit analysis for software reuse investment decisions:

1. Select the alternatives to analyze. Examples of cost-benefit alternatives for deciding on software reuse investments are:

   - Reuse parts from an identified source for a specific function within an application domain. This simple reuse alternative may lead to immediate benefits because software generally costs less to acquire than to develop. However, the organization may not have the necessary skills, the software may not exist, or existing software may take too long to acquire.
   - Redesign a specific function within an application domain for reuse within the organization in future applications. An organization with expertise in a particular application domain might repeatedly develop and maintain highly similar, but distinct versions of software. By creating a reusable component, the organization can relieve pressure to produce more function on a shorter development cycle and improve productivity.
   - Design a set of reusable parts for reuse outside of the organization. When the reuse consumers are other organizations within the same company, the benefits to the corporation motivate the investment. When the reuse consumers come from outside companies, revenue, fees, and royalties motivate the investment.

2. Determine the organization's priorities, goals, requirements, and business strategies that will influence the investment decision. These factors greatly influence the criteria for deciding whether the results of the cost-benefit analysis will indicate an invest or do not invest decision. An organization whose long-term business strategy is to develop software to obtain revenue from sales or licensing fees would likely invest more in building a library of reusable parts. An organization that provides custom software packages for a wide range of applications would more likely invest in obtaining and reusing parts from externally available reuse libraries. This means of reducing cost and shortening schedules especially applies to companies with limited resources.

3. Determine the time period for the analysis. It often takes more than the time frame of one project to realize the benefits of producing re-

Figure 2 Benefits and costs of producing reusable information

REUSE PRODUCER BENEFITS:		MEASUREMENT
b_1 Added revenue due to income from selling reusable information	$	× #users
b_2 Added revenue from fees or royalties resulting from the redistribution of information	$	× #users × #copies
REUSE PRODUCER COSTS:		
c_1 Cost of performing cost-benefit analysis	person months	× $/person month
c_2 Cost of performing domain analysis	person months	× $/person month
c_3 Cost of designing reusable parts	person months	× $/person month
c_4 Cost of modeling/design tools for reusable parts	$	
c_5 Cost of implementing reusable parts	person months	× $/person month
c_6 Cost of testing reusable parts	person months	× $/person month
c_7 Cost of documenting reusable parts (internal)	pages	× $/page
c_8 Cost of obtaining reuse library tools	$	
c_9 Cost of added equipment for reuse library	$	
c_{10} Cost of resources to maintain reuse library	person months	× $/person month
c_{11} Cost of management for development, test, and library support groups	person months	× $/person month
c_{12} Cost of producing publications	pages	× $/page
c_{13} Cost of maintaining reusable parts	person months	× $/person month
c_{14} Cost of marketing reusable parts	$	
c_{15} Cost of training in software reuse	$	

usable software because of the initial investments in analysis, design, tools, and library support. The cost-benefit analysis should span the time frame of the life of the application. For example, if an organization makes an initial investment in developing a library of objects that they expect to reuse in a set of applications spanning a three-year period, the cost-benefit analysis should span three years.

4. Identify and quantify the costs and benefits for each alternative. The list of costs and benefits varies, depending on the type and extent of the reuse investments considered. An organization must include costs associated with performing the cost-benefit analysis, obtaining reusable software, maintaining libraries, and developing tools. The organization must also take care not to bias the analysis results. In a typical cost-benefit analysis, organizations tend to overestimate the benefits and underestimate the costs.

5. Perform the cost-benefit analysis. Determine a break-even point, a payback period, and a list of intangible benefits for use in the analysis and final investment decision process.[4]

After performing the cost-benefit analysis and making the investment decisions, periodically revalidate the decisions and make adjustments to the investment areas. New information, changes in strategies and priorities, and initial investment results can affect how the organization should continue to direct its resources.

A cost-benefit analysis also requires an assumed discount rate (or time value of money). We include the time value of money because we must allow for the fact that we can invest today's money at an interest rate that makes it worth more tomorrow. Since cost-benefit analyses for software reuse generally span several years, we should use the time value of money for a more accurate analysis. The organization's business planner or financial analyst provides an acceptable discount rate. We then use the following equation for computing the net present value (NPV) of a particular investment over a time period of *0* through n years:[48]

$$\text{NPV} = \sum_{t=0}^{n} \frac{(B_t - C_t)}{(1+k)^t}$$

where k is the discount rate, B_t is the value of benefits in year t, and C_t is the value of costs in year t. Using the benefits and costs described in Figure 1 for the reuse of existing reusable information or Figure 2 for producing reusable information, we compute the values for B and C for each year in the cost-benefit analysis as follows:

$$B = \sum_{i=1}^{j} b_i, \quad C = \sum_{i=1}^{k} c_i$$

where j is the number of costs or benefits that apply to the investment within that year.

Using derived metrics to simplify cost-benefit analysis. We simplify the cost-benefit analysis procedure because of the solid foundation provided by the metrics. For example, we obtain DCA from the sum of development benefits in reduced cost to design, document, code, and test, minus integration and other development costs. SCA, in turn, represents the benefits of reduced software maintenance costs. To obtain the RCA metric we sum the DCA and SCA metric. As a result, the RCA metric quantifies the net benefits to a project that consumes reusable software.

The ADC metric accounts for the additional costs of producing reusable software. These include the costs of designing, testing, documenting, etc. Use of these metrics provides a simplified way to identify and quantify costs and benefits for reuse in step four of the cost-benefit procedure.

Project level ROI. In the absence of direct financial motivators such as fees, royalties, or dollar incentives, product managers are often reluctant to invest in a comprehensive reuse program, because the benefits of writing reusable code often accrue to projects outside their realm of responsibility. Therefore, any definition of return on investment should include benefits that other projects reap as a result of efforts by the initiating project. A straightforward formulation for return on investment includes the RCA and ADC metrics previously discussed:

$$\text{ROI} = \text{RCA} + \text{RCA}_0 - \text{ADC}$$

where

ROI = return on investment that occurs in infinite time

Figure 3 Business template program

```
SSI/CSI ..................... 80___ KLOC
RSI ......................... 20___ KLOC    __%
SSI/CSI written for reuse ..... 20___ KLOC    __%
Cost/LOC .................... $125__
Relative cost of reuse (RCR) .... .2___ (0-1)
Relative cost of writing reuse .. 1.5__ (1-2)
Error rate .................. 1.25_ Errors/KCSI
Cost/error .................. $20___ K

Data from other projects using your code:
Project SIRBO(K) Cost/LOC RCR Error/Rate Cost/Error
A       2        200      .2_  2          10
B       8        80       .3_  .5         18
        0        100      .2_             20
        0        100      .2_             20
        0        100      .2_             20
        0        100      .2_             20
        0        100      .2_             20
        0        100      .2_             20
        0        100      .2_             20
        0        100      .2_             20
```

Figure 4 Output from business template program

```
  SSI/CSI ..........................    80.00 KLOC
* Reuse percent ....................    20.00 %
  RSI ..............................    20.00 KLOC
  Percent SSI/CSI written for reuse .   25.00 %
  KLOC of SSI/CSI written for reuse .   20.00 KLOC
  Additional development cost (ADC) . $1250.00 K
  SIRBO ............................    10.00 KLOC
* Reuse value added (RVA) ..........     1.38
  Development cost avoidance (DCA) .. $2000.00 K
  Service cost avoidance (SCA) ...... $ 500.00 K
* Reuse cost avoidance (RCA) ........ $2500.00 K

  (Savings for other projects)
  Development cost avoidance (DCA0) . $ 768.00 K
  Service cost avoidance (SCA0) ..... $ 112.00 K
* Cost avoided by others (RCA0) ..... $ 880.00 K

  Total RCA (RCA + RCA0)............. $3380.00 K
→ ROI (RCA+RCA0-ADC) ..............  $2130.00 K
```

RCA = reuse cost avoidance for the initiating project

RCA_0 = reuse cost avoidance for other projects benefiting from the reusable code written by the initiating project

ADC = additional development cost to the initiating project of writing reusable code

Because individual projects usually have a limited duration, the ROI formula ignores the time value

of money. The new term, RCA_0, is similar to RCA in computation but we base it on SIRBO rather than RSI. We calculate RCA_0 by summing the RCA for each benefiting project as follows:

$$RCA_0 = \sum_{i=1}^{n} SIRBO_i \times (1 - \text{relative cost of reuse}_i) \times (\text{new code cost}_i) + \sum_{i=1}^{n} SIRBO_i \times \text{error rate}_i \times \text{cost per error}_i$$

where

$SIRBO_i$ = source instructions reused by project i
n = number of projects reusing code written by the initiating project
relative cost of $reuse_i$ = cost of integrating reusable code for project i relative to the cost of creating a new line of code, which is taken as 1
new cost $code_i$ = cost per line of code for project i
error $rate_i$ = number of errors per KLOC for project i
cost per $error_i$ = cost to repair an error for project i

To illustrate, assume an organization has an RCA of $2.5 million and an ADC of $1.25 million, as in the previous example. Projects A and B have already agreed to reuse some components and have the following data:

Project	SIRBO	Cost/ LOC	Relative cost	Error rate	Cost/ Error
A	2	200	0.2	2.0	10
B	8	80	0.3	0.5	18

Then

$$ROI = RCA + RCA_0 - ADC$$
$$= \$2.5 \text{ million} + RCA_0 - \$1.25 \text{ million}$$
$$= \$1.25 \text{ million} + (2 \text{ KLOC} \times 0.8 \times \$200 \text{ per line}) + (2 \text{ KLOC} \times 2 \text{ errors/KLOC} \times \$10\text{K per error}) + (8 \text{ KLOC} \times 0.7 \times \$80 \text{ per line}) + (8 \text{ KLOC} \times 0.5 \text{ error/KLOC} \times \$18\text{K per error})$$
$$= \$2.13 \text{ million}$$

Although not very complex, the number of computations make the final figure for ROI prone to error. For this reason, we wrote a business template program to automate the project level ROI analysis. The template provides defaults for many of the parameters previously described. However, the user can alter the default parameters to use actual project data or to simply experiment with different input values. Figure 3 shows a template with example data.

After the user enters the unique project-level data, the template program generates the output shown in Figure 4. The template program computes all reuse metrics in addition to providing the ROI for the project. The automation of the reuse metrics and ROI computations greatly assist the software project manager in developing justification to implement a reuse program at the project level.

Corporate-level ROI. A corporate-level reuse program may consist of many project-level reuse programs. In part, the costs and benefits that accrue to the corporation come from the sum of the costs (ADC) and benefits (RCA) to the individual projects. From a cost perspective, however, we must consider additional start-up activities. For example, a group of people might exist to promote reuse programs across the corporation. The corporation might fund tools to store, to search for, and to retrieve reusable parts. The reuse library may require a significant amount of disk storage to store the reusable parts. The corporation may decide to purchase parts from outside vendors rather than develop them locally.

These start-up activities may require a significant period of time before the reuse commitment be-

Figure 5 Example corporate-level ROI in thousands of dollars

Benefits/Costs	Year					
	Start-up	1	2	3	4	5
Benefits						
Total DCA	0	6,763	11,870	17,990	23,713	30,447
Total SCA	0	1,646	877	395	169	71
Total RCA	0	8,409	12,747	18,385	23,882	30,518
Support costs						
Reuse technology center	85	88	65	41	29	30
Site champions	241	500	598	717	858	1,024
Disk storage	0	182	321	481	621	779
Total support	326	770	984	1,239	1,508	1,833
Other costs						
Total ADC	0	3,730	5,299	6,247	6,009	5,008
Tool development	1,200	1,200	1,200	1,200	1,200	1,200
Vendor parts	2,400	1,450	1,450	1,450	1,450	1,450
Net savings per year	−3,926	1,259	3,814	8,249	13,715	21,027
Present value	−3,926	1,049	2,649	4,774	6,614	8,450
Net present value	19,610					
Internal rate of return	104 percent					

gins to yield productivity and quality savings. For this reason, a corporate-level ROI should take into account the time value of money. The most common way to express this ROI is through the net present value (NPV) approach, previously discussed, as follows:

$$\text{NPV} = -C_0 + \frac{B_1 - C_1}{1+k} + \frac{B_2 - C_2}{(1+k)^2} + \cdots + \frac{B_n - C_n}{(1+k)^n}$$

where

C_0 = corporate reuse start-up costs
B_i = benefits in year i
C_i = costs in year i
n = number of years for which revenues are to be considered
k = discount rate

Figure 5 displays an example of a corporate-level ROI. In this example, business planning practices dictate that we consider returns five years into the future. The hypothetical ROI is $20 million net present value with a 104 percent internal rate of return. Although this ROI seems extraordinarily high by conventional business standards, it does not reflect the risk inherent in many of the underlying assumptions of the ROI. For instance, we must make assumptions about the growth of reuse over time, the relative cost of reuse, the cost of writing reusable code, and the amount of vendor-purchased reusable code versus code written internally within the corporation. Because we base our assumptions on a range of probable outcomes, we should vary the assumptions to explore their effect on the ROI.

Notice that the ROI includes costs of tool development and support costs for persons who are either in a corporate reuse technology support center or who are site champions. In the IBM reuse program, these persons have the responsibility to communicate the benefits of reuse and to help spread reuse throughout the corporation and individual sites. Although individual projects do not incur these costs we must include them in the corporate ROI.

Concluding remarks

Software management depends on sound business decisions based on accurate measurements.[49] This paper introduces an investment model for software reuse[50,51] and the following new metrics: reuse percent, reuse cost avoid-

ance, additional development cost, and reuse value added. The metrics rely on easily collected data, provide reasonable representations of reuse activity, and encourage reuse. These metrics provide reliable input to the corporate reuse ROI model, where we carefully define the benefits attributed to reuse.

With emerging technologies, such as software reuse, we must extend the traditional role of metrics and ROI analysis. Metrics must not only assure the quality of reusable components, but they must also demonstrate the success of a program and improve the ability to plan and predict for future projects. Metrics also serve to encourage reuse by providing feedback on the results of a reuse program and by highlighting the benefits of an organizational reuse effort.

We made our ROI business template program available to all IBM sites to help convince project management to implement formal reuse programs. The IBM Reuse Technology Support Center also uses the corporate-level ROI to evaluate the benefit of formal reuse relative to other technologies that improve programmer productivity and quality.

We intend to continue to validate the measures and the ROI model. This includes comparing the predicted costs with actual costs avoided, and comparing increased productivity rates with the values calculated in the metrics and the ROI model. Although the model uses industry experience for default values in the equations, we use actual values when we have them. For example, we usually have actual software development costs and standard software development defect and maintenance data, and we routinely gather usage data on reusable components. We constantly compare and review these data with industry experience to maintain the accuracy of the model.

This paper discusses reuse measurements for software only. Future work will include methods to quantify reuse of information in areas other than software (e.g., design, test case, and information development). We want to capture, track, and validate our relative cost factors, such as the relative cost of developing reusable components. We also would like to study data related to the reuse process, such as the cost of certifying reusable components and the costs of maintaining our infrastructure of personnel and our reuse library.

Software reuse provides a promising answer to the challenges that confront most software organizations. A software organization that invests in software reuse can realize great improvements in productivity, cost reduction, and software quality. Faced with increasing demand for more function, reduced development cycle time, reduced development costs, and improved quality, the long-term competitiveness of any software development organization may depend on these improvements. To prepare, every organization should begin educating and involving its software developers in software reuse techniques and tools. Individuals skilled in the reuse techniques and the application domain can start by conducting an initial domain analysis. This domain analysis will provide the software organization with a set of software reuse alternatives appropriate to that organization's application and business environment.

Software reuse must become an integral part of the software development process. When this happens, software development will have evolved to the point where we find the development of commodities like hardware today. Only then can programmers keep pace with the demands of new hardware technologies and user requirements by spending resources on new, creative software rather than reworking and reinventing the old.

Acknowledgments

The authors thank the IBM Corporate Reuse Council for assistance with many of the concepts described in this paper and for supplying much of the data for examples. Also, we thank Mike Falcetano for his excellent work on the business template program and other related measurement tools.

Cited references

1. F. P. Brooks, "No Silver Bullet," *IEEE Computer* **4**, No. 2, 10–19 (April 1987).
2. C. W. Krueger, "Software Reuse," *ACM Computing Surveys* **24**, No. 2, 131–183 (June 1992).

3. B. T. Cox, "Planning the Software Industrial Revolution," *IEEE Software* **7**, No. 6, 25–33 (November 1990).
4. R. S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill Book Co., Inc., New York (1992).
5. R. S. Pressman and S. R. Herron, *Software Shock*, Dorset House, New York (1991).
6. W. Tracz, "Software Reuse: Motivators and Inhibitors," *Proceedings of COMPCON S'87, 1987*. Reprinted in W. Tracz, *Software Reuse: Emerging Technology*, IEEE Computer Society Press (1990), pp. 62–67.
7. W. B. Frakes, "Software Reuse, Quality, and Productivity," *Proceedings of International Software Quality Exchange*, Juran Institute Inc., Wilton, CT (March 1992), pp. 9–9 to 9–17.
8. T. C. Jones, "Reusability in Programming: A Survey of the State of the Art," *IEEE Transactions on Software Engineering* **SE-10**, No. 5, 488–494 (September 1984).
9. R. Prieto-Diaz, "Implementing Faceted Classification for Software Reuse," *Communications of the ACM* **34**, No. 5, 88–97 (May 1991).
10. P. Freeman, "Reusable Software Engineering: Concepts and Research Directions," *ITT Proceedings of the Workshop on Reusability Programming* (1983), pp. 129–137. Reprinted in P. Freeman, *Tutorial: Software Reusability*, IEEE Computer Society Press, New York (1987), pp. 10–23.
11. M. D. Lubars, "Wide-Spectrum Support for Software Reusability," *Proceedings of the Workshop on Software Reusability and Maintainability* (October 1987). Reprinted in W. Tracz, *Software Reuse: Engineering Technology*, IEEE Computer Society Press, New York (1990), pp. 275–281.
12. M. Lenz, H. A. Schmid, and P. Wolf, "Software Reuse Through Building Blocks," IEEE Software **4**, No. 4, 34–42 (July 1987).
13. T. Biggerstaff and C. Richter, "Reusability Framework, Assessment, and Directions," *IEEE Software* **4**, No 2, 41–49 (March 1987).
14. S. C. Bailin, R. H. Gattis, and W. Trusczkowski, "A Learning-Based Software Engineering Environment for Reusing Design Knowledge," *International Journal of Software Engineering and Knowledge Engineering* **1**, No. 4, 351–371 (December 1991).
15. R. Prieto-Diaz and G. A. Jones, "Breathing New Life into Old Software," *GTE Journal of Services and Technology*, Vol. 1. Reprinted in W. Tracz, *Software Reuse: Emerging Technology*, IEEE Computer Society Press, New York (1990), pp. 152–160.
16. J. M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering* **SE-10**, No. 5, 564–573 (September 1984). Reprinted in P. Freeman, *Tutorial: Software Reusability*, IEEE Computer Society Press (1987), pp. 181–191.
17. P. Freeman, "A Conceptual Analysis of the Draco Approach to Constructing Software Systems," *Tutorial: Software Reusability*, IEEE Computer Society Press, New York (1987), pp. 192–205.
18. B. H. Barnes and T. B. Bollinger, "Making Reuse Cost-Effective," *IEEE Software* **8**, No. 1, 13–24 (January 1991).
19. T. B. Barnes and S. L. Pfleeger, "Economics of Reuse: Issues and Alternatives," *Information Software Technology* **32**, No. 10, 643–652 (December 1990).
20. R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software* **4**, No. 1, 6–16 (January 1987).
21. W. Tracz, "Reusability Comes of Age," *IEEE Software*, 6–8 (July 1987).
22. R. Prieto-Diaz, "Domain Analysis for Reusability," *Proceedings of COMPSAC '87*, IEEE Computer Society Press, New York (1987), pp. 23–29.
23. G. Booch, *Object Oriented Design with Applications*, Benjamin/Cummings, Redwood City, CA, 1991.
24. M. A. Simos, "The Domain-Oriented Software Life Cycle: Towards an Extended Process Model for Reusability," *Proceedings of the Workshop on Software Reusability and Maintainability* (October 1987). Reprinted in W. Tracz, "Software Reuse: Emerging Technology," IEEE Computer Society Press (1990), pp. 354–363.
25. B. Meyer, "Reusability: The Case for Object-Oriented Design" *IEEE Software* **4**, No. 2, 50–64 (March 1987).
26. B. A. Burton, R. W. Aragon, S. A. Bailey, K. D. Koehler, and L. A. Mayes, "The Reusable Software Library," *IEEE Software* **4**, No 3, 25–33 (July 1987).
27. G. M. Bowen, "An Organized, Devoted, Project-Wide Reuse Effort," *Ada Letters* **12**, No. 1, 43–52 (January/February 1992).
28. T. Standish, "An Essay on Software Reuse," *IEEE Transactions on Software Engineering* **10**, No. 5, 494–497 (1984).
29. R. D. Banker and R. J. Kauffman, "Reuse and Productivity in Integrated Computer Aided Software Engineering: An Empirical Study," *MIS Quarterly*, 375–401 (September 1991).
30. *Repository Guidelines for the Software Technology for Adaptable, Reliable Systems (STARS) Program*, Contract No. F19628-88-D-0032, CDRL No. 0460, Technical Report Center, Morgantown, WV 26505 (March 15, 1989).
31. W. M. Thomas, A. Delis, and V. R. Basili, "An Evaluation of Ada Source Code Reuse," *Proceedings of 11th Ada Europe International Conference*, Zandvoort, Netherlands (June 4–5, 1992), pp. 80–91.
32. J. Margano and L. Lindsey, "Software Reuse in the Air Traffic Control Advanced Automation System," paper for the *Joint Symposia and Workshops: Improving the Software Process and Competitive Position*, Alexandria, VA (April 29–May 3, 1991).
33. D. G. Firesmith, "Managing Ada Projects: The People Issues," *Proceedings of TRI Ada '88*, Charleston, WV (October 24–27, 1988), pp. 610–619.
34. C. Jones, *Applied Software Measurement: Assuring Productivity and Quality*, McGraw-Hill, Inc., New York (1991).
35. J. E. Gaffney, Jr. and T. Durek, *Software Reuse—Key to Enhanced Productivity; Some Quantitative Models*," Software Productivity Consortium, *SPC-TR-88-015* George Mason University, Center for Software and Systems Engineering, Herndon, VA (April 1988).
36. J. E. Gaffney, Jr. and T. A. Durek, "Software Reuse—Key to Enhanced Productivity: Some Quantitative Models," *Information and Software Technology* **31:5** (June 1989).
37. *Standard for Management of a Reusable Software Component Library*, NATO Communications and Information Systems Agency (August 18, 1991).
38. *Standard for the Development of Reusable Software Components*, NATO Communications and Information Systems Agency (August 18, 1991).
39. J. E. Gaffney, Jr. and R. D. Cruickshank, "A General

Economics Model of Software Reuse," *Association for Computing Machinery*, Melbourne, Australia (May 1992).
40. V. R. Basili and R. W. Selby, "Paradigms for Experimentation and Empirical Studies in Software Engineering," *Reliability Engineering and System Safety* **32**, 171–191 (1991).
41. J. S. Poulin and J. M. Caruso, "A Reuse Measurement and Return on Investment Model," *Proceedings of the Second International Workshop on Software Reusability*, Lucca, Italy (March 24–26, 1993), pp. 152–166.
42. J. S. Poulin, "Issues in the Development and Application of Reuse Metrics," *Fifth International Conference on Software Engineering and Knowledge Engineering* (June 16–18, 1993), pp. 258–262.
43. M. K. Daskalantonakis, "A Practical View of Software Measurement and Implementation Experiences Within Motorola," *IEEE Transactions on Software Engineering* **18**, No. 11, 998–1010 (November 1992).
44. S. Woodfield, D. Embley, and D. Scott, "Can Programmers Reuse Software?," *IEEE Software* **4**, No. 4, 52–53 (July 1987).
45. W. Tracz, "Software Reuse Myths," *ACM SIGSOFT Software Engineering Notes* **13**, No. 1, 17–21 (January 1988).
46. J. Favaro, "What Price Reusability? A Case Study," *Ada Letters* **11**, No. 3, 115–24 (Spring 1991).
47. *Software Engineering Strategies*, Strategic Analysis Report, Gartner Group, Inc., Stamford, CT (April 30, 1991).
48. J. L. King and E. L. Schrems, "Cost-Benefit Analysis in Information Systems Development and Operation," *ACM Computing Surveys* **10**, No. 1, 19–34 (March 1978).
49. A. J. Albrecht, "Measuring Application Development Productivity," in *Proceedings of the Joint IM/SHARE/GUIDE Application Development Symposium* (October 1979), pp. 83–92.
50. D. J. Reifer, "Reuse Metrics and Measurement—A Framework," NASA/Goddard Fifteenth Annual Software Engineering Workshop (November 28, 1990).
51. J. S. Poulin and J. M. Caruso, "Determining the Value of a Corporate Reuse Program," *Proceedings of the IEEE Computer Society International Software Metrics Symposium*, Baltimore, MD (May 21–22, 1993), pp. 16–27.

Accepted for publication February 25, 1993.

Jeffrey S. Poulin *IBM Federal Systems Company, Owego, New York 13827.* Dr. Poulin joined IBM's Reuse Technology Support Center in Poughkeepsie, New York, in 1991, where his responsibilities included developing and applying corporate standards for reusable component classification, certification, and measurements. As a member of the RTSC, Dr. Poulin helped lead the development and acceptance of the IBM software reuse metrics and return on investment model. Dr. Poulin currently works as an advisory programmer with the IBM Federal Systems Company Open Systems Development group on software reuse and systems integration issues. He participates in the IBM Corporate Reuse Council, the Association for Computing Machinery, and the IEEE Computer Society. A Hertz Foundation Fellow, Dr. Poulin earned his bachelor's degree at the United States Military Academy at West Point, New York, and his master's and Ph.D. degrees at Rensselaer Polytechnic Institute in Troy, New York.

Joseph M. Caruso *IBM Large Scale Computing Division, Poughkeepsie, New York 12601.* Dr. Caruso is an advisory systems analyst whose responsibilities include the determination of return on investment for the many technologies funded by his organization. He joined IBM in 1978 as a systems analyst for Material Requirements Planning Systems. In 1985 he moved into the assurance department to pursue his interests in statistics. While in that department, he developed software statistical tools, projected quality levels of software projects, automated reporting and analysis systems, and provided statistical education for the quality assurance organization. His interests include applications of software reliability growth modeling, sample size determination, and Monte Carlo simulation. He received his B.S. degree from the State University of New York at Stony Brook, his M.S. degree from Pennsylvania State University, and his Ph.D. degree from Union College, New York.

Debera R. Hancock *IBM PC Company Technology Center, Boca Raton, Florida 33487.* Ms. Hancock is a senior programmer and manager whose primary responsibility is the management of software design and development projects for IBM PS/2 systems. She has seven years experience managing IBM software and microcode development projects to support IBM midrange and personal systems. She is interested in object-oriented design and software reuse as a method of improving programming productivity and product quality. Ms. Hancock is currently the engineering software representative to the IBM Corporate Reuse Council and is a member of the Association for Computing Machinery. Prior to joining IBM, she was a systems programmer for large systems operating systems for the Burroughs Corporation and programmed machine tool and robotics applications for the General Electric Company. Ms. Hancock received her B.S. degree from the University of Delaware, and is currently pursuing an M.S. degree in computer engineering at Florida Atlantic University, Boca Raton, Florida.

Reprint Order No. G321-5525.