Routledge
Taylor & Francis Group

Journal of Management Information Systems

ISSN: 0742-1222 (Print) 1557-928X (Online) Journal homepage: http://www.tandfonline.com/loi/mmis20

Software Reuse: Survey and Research Directions

Yongbeom Kim & Edward A. Stohr

To cite this article: Yongbeom Kim & Edward A. Stohr (1998) Software Reuse: Survey and Research Directions, Journal of Management Information Systems, 14:4, 113-147, DOI: 10.1080/07421222.1998.11518188

To link to this article: http://dx.doi.org/10.1080/07421222.1998.11518188

Software Reuse: Survey and Research Directions

YONGBEOM KIM AND EDWARD A. STOHR

YONGBEOM KIM is an Assistant Professor of Information Systems at Fairleigh Dickinson University. He received his Ph.D. in information systems from New York University and a B.S. and M.S. in electrical engineering from Seoul National University, Korea. His current research interests include software reuse, performance evaluation of interactive computing systems, and information systems security.

EDWARD A. STOHR holds a Bachelor of Civil Engineering degree from Melbourne University, Australia, and an M.B.A. and Ph.D. in information science from the University of California, Berkeley. Prior to coming to Stern School of Business at New York University in 1979, he was an Associate Professor at the Graduate School of Management, Northwestern University. From 1984 to 1995 he served as Chairman of the Information Systems Department at Stern. Currently, he is Director of the Center for Research in Information Systems at the Stern School. Professor Stohr's research interests center on the use of information technology to support decision making in organizations.

ABSTRACT: Software reuse is the use of software resources from all stages of the software development process in new applications. Given the high cost and difficulty of developing high-quality software, the idea of capitalizing on previous software investments is appealing. However, software reuse has not been as effective as expected and has not been very broadly or systematically used in industry. This paper surveys recent software-reuse research using a framework that helps identify and organize the many factors that must be considered to achieve the benefits of software reuse in practice. We argue that software reuse needs to be viewed in the context of a total systems approach that addresses a broad range of technical, economic, managerial, organizational, and legal issues and conclude with a summary of the major research issues in each of these areas.

KEY WORDS AND PHRASES: reuse metrics, reuse process, reuse technologies, software reuse.

U.S. BUSINESS CURRENTLY SPENDS OVER $400 BILLION PER YEAR or almost 50 percent of all new capital budgeting expenditures on information technology [91]. These expenditures have not always resulted in clear gains in productivity; some IT investments are successful while others are economic failures [90]. The emerging business environment is characterized by increased competition, global markets, and the pressure to cut costs. This makes the need for successful investments in IT more acute.

Journal of Management Information Systems / Spring 1998, Vol. 14, No. 4, pp. 113–147

0742–1222 / 1998 $9.50 + 0.00.

Unfortunately, the process of software development is plagued by cost overruns, delayed schedules, unsatisfied requirements, and a shortage of competent software professionals.

Software reuse is the use of previously developed software resources in new applications. Because fewer total lines of code need to be written, software reuse can increase productivity, reduce development costs, and minimize schedule overruns. Since reusable software resources should be rigorously tested and verified, reuse also has the potential to improve software quality, reliability, maintainability, and portability. Typically, software reuse involves the reuse of portions of code (e.g., library subroutines) by other programmers in the same organization. In this paper we are concerned with a broader concept of reuse in which all of the products of the software life cycle are reused by developers within the same as well as different organizations on a broad spectrum of development tasks and across a variety of software application domains. We refer to this as "widespread software reuse."

Software reuse has been the subject of numerous articles and books [8, 39, 48, 86]. In addition, there are a number of reports of successful industry experience [35, 38, 53, 60]. Despite these successes, software reuse has had limited acceptance in industry [1, 21]. Barriers to achieving the promise of software reuse include the paucity of tools to help catalog, refine, and compose resources [31, 76], the lack of a formal representation for reusable software resources [12, 81], and the lack of managerial and organizational support [35, 64].

We survey existing research on software reuse using a framework that encompasses a broad range of technical and nontechnical issues. The objective is to suggest research directions for future studies that will examine often-overlooked issues and have a cumulative impact on our understanding of software reuse. Our basic argument is that economic, technical, behavioral, and organizational issues must all be addressed if software reuse programs are to be successful.

A Framework for Software Reuse

FRAMEWORKS CAN BE USED TO ORGANIZE THE EXISTING LITERATURE on software reuse, to evaluate trends, and to point to areas needing further study. A number of conceptual frameworks for research on software reuse have been proposed in the literature. These frameworks fall into two groups: one that emphasizes the different types of reusable resource and another that focuses on the different technologies to achieve reuse.

In the former group, Freeman [32] developed a framework based on three dimensions: the artifact being reused, the manner in which it is reused, and the factors needed to enable successful reuse. For the first dimension, five levels of reusable information were distinguished: code fragments, logical structure (process and data architectures), functional architecture (user-oriented specifications of functions performed), external knowledge (software development knowledge and application domain knowledge), and environmental knowledge (technology transfer and how software is utilized in the organization). Note the progression of resource types from concrete to abstract, from

specific to general, and from later stages of the life cycle to earlier stages. In the same vein, Isoda [41] defines three types of software reuse based on the life-cycle stage of the resource: (1) final products such as program code, requirements specifications, design, and test studies, (2) process information such as design rationales, and (3) application domain knowledge. Data, test cases, and documentation are additional types of reusable resource that are included in some frameworks [44].

Most existing frameworks belong to the second group; they organize the discussion around the different techniques for achieving software reuse. Frameworks in this class, for example, Biggerstaff and Richter [9], usually distinguish two major approaches: composition technologies and generation technologies. Composition technologies involve the reuse of software building blocks such as code fragments, subroutines, and objects. Generation technologies involve the reuse of programming constructs via high-level languages or application generators. Using a finer partitioning scheme based on the level of "abstraction," Krueger [52] divides the technologies of software reuse into eight categories: high-level languages, design and code scavenging, source code components, software schemes, application generators, very high-level languages, transformational systems, and software architectures. Each software reuse approach is characterized in terms of its reusable resources and the way they are abstracted, selected, specialized, and integrated into the target system.

The above frameworks have been developed to introduce a particular perspective on software reuse, or to organize technical approaches to enhancing software reusability. They are less appropriate for explaining the broad range of issues that determine the success or failure of software reuse programs in industry. To cover the many economic, technical, and nontechnical issues that have an impact on the effectiveness of software reuse, we therefore use the framework in Table 1 to organize our discussion. There are three categories of research issues in the framework: general issues (definition and scope of software reuse and economic issues), technical issues (software reuse process and software reuse technologies), and nontechnical issues (behavioral, organizational, and legal and contractual issues). Examples of research addressing the impediments to successful reuse are included in the third column of Table 1. The following sections discuss each of the research issues in the second column of Table 1.

Definition and Scope of Software Reuse

QUESTIONS RELATED TO THE DEFINITION AND SCOPE OF SOFTWARE REUSE have been examined by a number of researchers [41, 81, 94]. As in any new field, the terminology of reuse is evolving. In this section we provide several definitions on which there is general agreement. As discussed later, the terminology becomes less standard when the details of reuse techniques and metrics for measuring the costs and benefits of reuse are considered.

Software reuse is generally defined as the use of previously developed software resources from all phases of the software life cycle in new applications by various users such as programmers and systems analysts [13, 52]. A *reusable resource* can be

Table 1. A Framework for Software Reuse Research

Category	Research issues	Impediments to software reuse
General issues	Definition and scope	The lack of well understood and accepted terminology to describe concepts [76].
	Economic issues	The investment needed to promote software reuse [1, 39]; the lack of an economic model to explain the benefits and costs of software reuse [81].
Technical issues	Software reuse process	The lack of a methodology for creating and implementing software reuse [1].
	Software reuse technologies	The lack of reusable and reliable software resources [17, 77]; the lack of tools and techniques for supporting software reuse [18, 21].
Nontechnical issues	Behavioral issues	The lack of commitment, encouragement, training, and rewards for software reuse [1]. The NIH (not invented here) syndrome [97].
	Organizational issues	The lack of organizational support to institutionalize software reuse [63, 81]; the difficulty in measuring the gains from reuse [48].
	Legal and contractual issues	Intellectual property rights and contractual problems of software reuse [21].

any information in physical or electronic form that a developer may need in the process of creating software [32]. *Reusability* is a measure of the ease with which the resource can be reused in a new situation. Some classes of resource are naturally more reusable than others. *Reuse* occurs when a developer (consumer or client) uses a resource developed by another software developer (producer or donor.) The distinction between consumption and production of reusable resources is also captured by the terms "development with reuse" and "development for reuse." Software reuse may be *ad hoc or opportunistic* in the sense that developers discover reusable components in existing applications by a process commonly termed "code scavenging." On the other hand, *planned reuse* occurs when an organization develops explicit reuse processes and standards and, in particular, invests in the up-front development of reusable resources.

In practice, most reuse has involved the reuse of code by developers working on a common project [6, 7]. However, this is limiting. A more ambitious program of reuse presents greater challenges but can have major benefits. To provide an organized and inclusive point of view, we define the concept of widespread software reuse with

Classes of User:	Original Developers, Other developers, Other organizations
Resource Types:	
Entity Types:	Processes, Objects, Data, Test cases, Documentation
Abstraction Levels:	Concrete (code, objects), Design artifacts, Abstract knowledge
Application Types:	Customized, Functional, Generic,
Development Task:	Maintenance, New projects /related domain, New projects/different domain

Figure 1. Dimensions of Widespread Software Reuse

respect to the following dimensions: classes of user, reusable resource types, and software development tasks (see figure 1).

There are issues with regard to each of the dimensions in figure 1 which we outline here and discuss more fully later. The issue with regard to user classes is that great difficulty has been experienced in extending reuse beyond a single developer reusing his or her own software resources [18, 81]. Obviously, significant benefits can only be obtained from reuse of software resources by others, and, for organizations such as the Department of Defense that employ many software contractors, reuse across different organizations is essential [21].

The next set of issues concerns what can be feasibly and economically reused. Software resources can be classified according to entity type, level of abstraction (or stage in the development life cycle in which they are produced), and application type. By "entities" we mean the fundamental things that comprise software resources. The most common reusable software entity types are processes, data, and objects. Test cases (consisting of data and procedures) and documentation (plans, estimates, user manuals, and so on) are other major classes of software resource that can be reused in many situations with obvious cost savings. To a large extent, data, test case, and documentation reuse can be gained by a mixture of organizational discipline and the use of some relatively mature technologies such as data dictionaries, database management systems, and version control software. Because they present a more challenging and difficult problem, process resources have been the major target of reuse research and will be the main focus of this paper.

Software development can be viewed as a process in which abstract software resources are continually changed into more concrete forms. For example, a process resource is near one end of the spectrum, the abstract level, if it is represented by functional requirements in narrative form. A process resource is near the opposite end of the spectrum, the concrete level, if it is in a form that can be directly used in a functioning software system, such as object or source code. In the middle of the spectrum, one has software resources represented in data-flow diagrams, entity-relationship diagrams, system architectures, and design rationales. While most reuse has been concerned with concrete resources, many researchers contend that larger savings can be obtained by the reuse of more abstract resources [7]. This is because the cost of coding is generally only a small portion of the total software development cost [11].

At the most abstract (but possibly most important) level, experienced designers and programmers reuse internal "informal" software resources such as design and coding rationales learned from experience. The difficulty is in gathering and representing this tacit, implicit knowledge so that others may reuse it [9].

Software resources occur in a wide range of contexts on a continuum from customized resources to functional resources to generic resources. A *customized* process resource is a set of application functions developed to satisfy the specific requirements of users in an organization. Common examples are data-processing systems for payroll and accounts receivable. A *functional* process resource is a set of application functions that are packaged as a unit for a given application area such as management science, finance, accounting, or statistics. Each function of a functional collection can be used separately. Mathematical packages such as IMSL [40] fit this category. Recently, comprehensive "enterprise-wide" packages such as the SAP R/3 system are becoming popular in manufacturing and financial applications [15]. A *generic* process resource is a general-purpose software resource that can be used in many different application domains. Examples include file management, screen management, graphics, and date manipulation routines. Customized and functional resources are sometimes referred to as "vertical" resources because they apply in a given domain, while generic resources are called "horizontal" resources because they apply across different domains [9].

The final set of issues involves the type of software task, which may vary from maintaining existing systems to developing new software. Maintenance can be viewed as a reuse-oriented task in which the appropriate requirements, design, code, and test data from earlier versions of the system must be accessed and understood by the maintenance programmer. Because understanding of the existing system can be so difficult, abstract software resources from early phases in the life cycle can be as important as the code itself in maintenance tasks. The main business issue with regard to the development of new software concerns the relative novelty of the application. When a series of future applications in a stable domain is contemplated, it is worthwhile investing in an extensive domain analysis (see later) and developing software libraries and management procedures as has been done in the Japanese "software factories" [24]. On the other hand, for a "one-off" system in a new domain, planned reuse will be relatively unattractive. In this case, reuse may be limited to generic resources such as report writers, screen managers and date routines.

In summary, one of the problems of assessing the impact of software reuse in practice is that there are so many definitions and interpretations of what constitutes reuse. In this section, we have attempted to clarify this issue by analyzing the various dimensions of reuse. While figure 1 emphasizes the breadth of reuse possibilities, most research on software reuse has concentrated only on the reuse within a single firm of concrete process resources in new projects. There is a need to support the reuse of a broad range of software resources by various software developers in a wide range of tasks from maintenance of existing systems to development of new systems.

A useful line of research would be to determine the marginal returns that an organization can gain by investing in each of the different classes of reusable software

resource, and on how reuse techniques can improve productivity in each type of software development task. The value of investing in a reuse program is likely to vary according to the portfolio of project types that the firm has to undertake. For example, a firm involved primarily in maintenance activities might direct its energies to perfecting means for representing customized resources of all types, while a firm that is completely restructuring itself around new systems over a long time horizon might be better served by a reuse program involving primarily generic and functional resources. While the preceding statements seem plausible, the relationship between the overall IT development strategy and reuse strategy is complex and could present a fruitful area for research.

Economic Issues

THE REUSE OF SOFTWARE RESOURCES CAN RESULT IN INCREASES in productivity, reduced development lead times, improved software quality, and greater maintainability [12]. However, the implementation of a reuse program also has organizational implications and costs. In this section we discuss the benefits and costs of software reuse and review several cost-benefit models and project cost estimation models.

Benefits and Costs

A successful program of software reuse provides benefits in three areas: increased productivity and timeliness in the software development process, improved quality of the software product, and an increase in the overall effectiveness of the software development process.

Software productivity is measured in lines of code (LOC) or function points (FP) produced per dollar or per unit of time (usually person-months) [55]. There is little rigorous empirical evidence to support productivity increases from reuse [87] and a great deal of variance in the way productivity is measured. Interpretation of published reuse productivity data is also complicated because of contextual factors such as expertise, complexity of the application domain, and the use of quality control techniques, which vary from organization to organization. However, the reported results from many software reuse projects show increases in productivity with reuse [21, 29] and productivity increases on the order of 20 to 40 percent seem common [73]. For example, the average programmer productivity in a Japanese "software factory" was 500 to 800 source lines of code (LOC) per month without reuse and 800 to 3200 LOC per month with reuse [60].[1] Productivity should improve with increasing rates of reuse. Standards for measuring the reuse of abstract resources have not yet been developed. For concrete software resources, the reuse rate is defined as the proportion of previously developed code to new code where the proportion is usually measured in lines of code [54]. Reuse rates as high as 40 percent to 60 percent have been reported [29, 60, 97]. The economics of reuse also depends on the number of times that reusable resources are reused throughout the organization. For example, simple reusable components can show a payoff if they are reused twice, while the most

complex components may require from six to thirteen reuses [67].

The increase in software quality with reuse comes about because reusable components are usually subjected to rigorous testing and, over time, become almost defect-free. While quality can be measured in different ways, the usual measure is in terms of defects per LOC or FP discovered in a specified exposure period. Using reusability (and other) techniques, the reliability in the Fuchu Software Factory was two to three faults per 1,000 source lines [60], while one defect per 1,000 lines of code was reported in NASA's third Ada project, which also used reusability techniques [21].[2]

Other attributes of high-quality software such as understandability, adaptability, and portability can also be improved by high levels of reuse. All three attributes depend on how well the code can be generalized and on the quality of the documentation. Understandability of the software is increased by the use of familiar, well-documented standardized components from a reuse library. Adaptability refers to the extent to which a software component can be changed to satisfy similar, but different, requirements. Portability refers to the extent to which a software component can be used in multiple machine environments (the physical hardware, operating system, run-time environment, and compiler conventions). Adaptability and portability are needed to achieve reusability across multiple application and execution environments [16]. Portability depends also on the development of standards—either within the firm or, more generally, within the software industry.

The consistency and manageability of the overall process of software development is another area of benefit from software reuse. Consistency is improved by the reuse of the same software resources in many applications. Manageability is improved by the resulting improvements in project estimation and control procedures. Most important, the various attributes of high-quality software discussed above can contribute to a reduction in maintenance costs, which contribute over 60 percent of total software costs in many organizations [11]. Finally, software reuse can lead to new business opportunities such as selling reusable software resources and attacking niche software markets through cost reduction and differentiation [37].

A number of costs are associated with high levels of software reuse. The first set of costs occur on the consumer side in the reuse process itself. Reuse costs are the costs needed to find, understand, adapt, and integrate resources into the final software product. The second set of costs, investments in software reuse, can only be paid off from future projects after multiple uses of reusable software resources. A reuse program requires an initial investment in the development of a reuse library and the acquisition of reuse tools for cataloging, retrieving, and modifying reusable software resources. Other necessary initial investments are required to develop new organization structures and controls, to train the developers and users of reusable software resources, and to develop a culture where reuse can be successful [1, 73]. Ongoing investments are required to maintain the reuse library and develop software resources for reuse. For example, the effort to generalize, test, certify, and document a software resource and store it in the library may cost up to 200 percent of the normal cost of the component [97].

A reuse program is cost-effective when the accumulated difference between development

costs without reuse and development costs with reuse is greater than the investment for software reuse. Because these investment costs are generally not within the budget or schedule of a single software project, software developers and managers who are concerned more with short-term payoffs may be reluctant to support software reuse. To encourage reuse and to ensure proper evaluation in terms understood by management, research is needed on how to institutionalize the idea that reuse is an organization-wide investment [5] and that reusable software resources are financial assets [36].

Cost-Benefit Models and Reuse Metrics

The central idea of reuse cost-benefit models is to develop mathematical relationships that express the benefits and costs of reuse in terms of metrics that can be captured in a software development organization. A typical model was developed by Gaffney and Durek [33] and is briefly described here. Let:

C = cost of software development with reuse for a given product relative to the cost of the same product if it were built with all new code (for which $C = 1$);
R = reuse rate or proportion of reused code in the product ($R < 1$);
b = cost, relative to that for developing new code, of incorporating reused code into the new product ($b < 1$);
E = the relative cost of creating reusable code compared with normal code (presumably, $E > 1$); and
N = the number of the number of uses over which the reusable code cost is to be amortized.

Then:

$$C = (1 - R) * 1 + R * b + R * (E/N). \tag{1}$$

The three components in this formula represent, respectively, the cost to develop the new code, the cost of incorporating reusable code into the product, and the cost to develop the reusable component amortized over N reuses. As an example, using typical values achieved in the literature: $R = 0.3$, $b = 0.2$, $E = 1.5$ [73], and, assuming $N = 5$, the value of $C = 0.85$. Obviously, high values of reuse proportion, R, and a larger number of reuses, N, over which the development cost is amortized are desirable to achieve high productivity gains. It is also important to minimize E and b which represent, respectively, the costs of developing reusable software components and integrating them into the final software product. Setting $C = 1$ in (1), the minimum value for number of reuses that could make reuse economical is given by: $N_0 = E/(1 - b)$. Using the above values of E and b, $N_0 = 2$. Models similar to equation (1) were also developed in [33] for gauging the quality benefits (due to fewer software errors) and time benefits of software reuse.

Note that the productivity and other benefits cited in the previous section were measured from historic records by comparing projects or organizations that used reuse

methods with those that did not. The reuse model (1), however, adopts a cost-avoidance philosophy—the benefits are computed relative to what would occur if there were no reuse and all code had to be written from scratch. This approach has been used to develop a multiperiod return-on-investment model to determine if the initial investment in a software reuse program is worthwhile [73]. The basic idea is to convert (1) to dollar terms by multiplying each term by estimates of the software size (in LOC) and development cost per LOC, and to add a term that estimates the maintenance benefits from the reduced error rates obtainable with reuse. A different class of model is needed to develop cost estimates for individual projects. One approach is to modify the well-known COCOMO software cost estimation model [11] to allow for different types of reuse, the differential costs of developing for and with reuse, and the stage of the development life cycle in which reuse occurs [3]. Later we discuss another application of reuse modeling in the area of ongoing assessment of reuse programs.

While the above discussion has focused on metrics based on lines of code, it is not clear if this is a suitable basis for measurement in CASE or object-oriented environments. For example, Banker et al. [4] describe metrics for evaluating the effectiveness of CASE tool reuse repositories in terms of the software objects produced and reused rather than in terms of LOC or FP.

Summary: Benefits and Costs of Reuse

The claimed benefits from reuse are impressive and the reported results on reuse programs within individual commercial and government organizations have generally been positive. Theoretical studies using economic models such as the above support the idea that software reuse can be a keystone in efforts to improve productivity. They also suggest ways to standardize measurement methods, to justify reuse investments, and to encourage software reuse. However, neither the reported results nor the models unequivocally confirm the advantages of reuse programs. Nor do they provide adequate information on the conditions for success or failure. There is a need for studies that investigate reuse across a broad spectrum of companies and that relate reuse practices to achieved results using a standard set of terms and a common cost-benefit model. Particular issues that need investigation include:

- Metrics for the reuse of "abstract" software resources (such as analysis and design artifacts) and methods to estimate the benefits of reusing of such resources;
- Metrics for reuse in an object-oriented environment including the reuse of objects and class hierarchies and the effectiveness of inheritance;
- The relative cost-effectiveness of various development technologies such as application generators, very high level languages, CASE tools, and object-oriented languages.

Finally, more comprehensive economic models of software reuse that account for the large capital investment involved, cover a broad range of types of software resources, and consider software development as an ongoing activity are needed to

help investment decisions, address measurement problems, and provide analyses of software reuse approaches.

The Software Reuse Process

THIS SECTION ADDRESSES THE SECOND SET OF REUSE ISSUES IN FIGURE 1, beginning with a consideration of how reuse activities can be integrated into the systems development life cycle. Although programmers may informally reuse existing software resources in new applications, traditional software development methodologies (such as the software development life cycle and prototyping) do not explicitly support software development from reusable resources [70, 73]. Software reuse does not happen by accident [94]. Reusable resources must be intentionally designed and developed, and users need to know of their existence before being able to reuse them [81]. In this section we present a software reuse process that considers both activities for producing reusable software resources and activities for consuming them. We then describe a model of reuse-based software development that integrates these reuse activities into the software development life cycle. The latter model will be used as a framework for our discussion of reuse technologies in the subsequent section.

The software reuse process is shown in Table 2, with reuse activities divided into two groups: producing activities which involve the identification, classification, and cataloging of software resources, and consuming activities which involve the retrieval, understanding, modification, and integration of these resources into the software product. The third column of Table 2 cites some relevant research on each of these reuse activities.

A model of reuse-based software development that includes the above reuse activities is shown in the form of a data-flow diagram in figure 2. Eight steps (distinct activities) are represented by the bubbles. The first two steps involve the production of reusable software resources. The first step is to analyze existing software resources (developed internally or externally) to *identify* those that are suitable for reuse. The second step is to *classify and catalog* the identified reusable software resources. These two steps have to be performed at the initiation of the reuse program to develop a library of software resources, as well as each time a new software resource is acquired.

The remaining steps involve the consumption of reusable software resources and the development of new software resources. Step 3, *specifying* requirements for the new system, has to be performed regardless of whether or not the software resource is to be developed from scratch. Step 4, *retrieving* appropriate reusable software resources from the software library, is unique to the software reuse approach. The fifth step is to *understand* and assess the functionality of the selected resources in order to use or modify them. The sixth step, *modifying* software resources, is necessary when the retrieved software resources do not exactly match the requirements specification. The seventh step, *building* new software resources, is necessary when there are no similar software resources in the software library for some of the requirements. Finally, the eighth step is required to *integrate* the new and reusable software resources

Table 2. The Software Reuse Process

Activity category	Activities	Similar activities proposed by other researchers
Producing reusable resources	Identification	Decomposition/abstraction [12], Abstraction/analysis/ selection/generalization [34]
	Classification	Classification [12], Cataloging/attribution [34]
Consuming reusable resources	Retrieval	Selection [12, 79]
	Understanding	Understanding [9]
	Modification	Specialization/adaptation [12, 34, 79], Modifying [9]
	Integration	Composition [9, 12], Assembly [79], Interconnection [34]

into the target software system. Note that normal systems development methodologies involve only steps 3, 7, and 8 in figure 2. For any software project, the benefit of the software reuse approach is given by the difference between the cost of performing step 7 for the case when all resources are developed from scratch and the costs of performing step 7 for some (new) resources and steps 4, 5, and 6 for other (reusable) resources.

Since we are dealing with reusable resources from all software development activities, the reusable resources in the software library in figure 2 can be in the form of specifications, data-flow diagrams, program structure charts, objects, source code, or object code. If a strict life-cycle approach is used, the steps in figure 2 might be iterated across all relevant reusable resources for each phase of the life cycle in order to complete the specification for the entire target system (or a subsystem of the target system) one phase at a time. Alternatively, in an evolutionary or prototyping approach to systems development, the user might wish to retrieve the documentation for all life-cycle phases for a single reusable resource, make any relevant changes, and then include the resource in the prototype. Similarly, in maintenance mode, the user will probably wish to look at the documentation from all phases (levels of abstraction) for the resources that need to be modified.

The steps in figure 2 can be performed independently of the use of CASE tools or application generators. The advantage of CASE tools for reuse is the existence of a repository of software-related artifacts that record domain knowledge and are linked together through all the stages of the software development life cycle [47]. In contrast, most current reuse technologies support only independent single resource reuse at the coding phase. To support reuse, a traditional CASE tool could be extended to provide automated support for all the processes shown in figure 2.

To summarize this section, we have described a reuse-based software development model that integrates activities for producing and consuming reusable resources into the normal software development cycle. Research is needed on the technical and

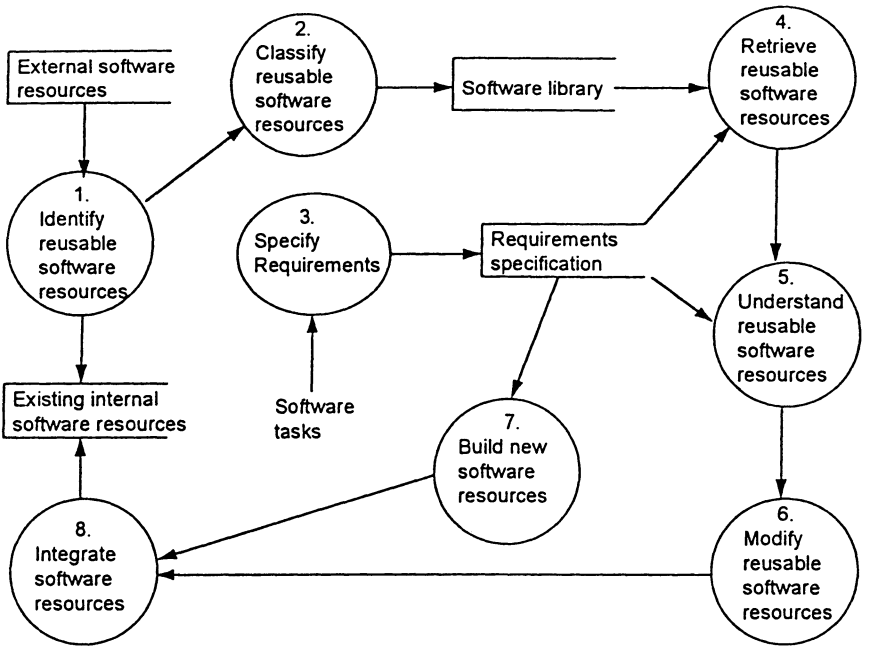

Figure 2. A Model of Reuse-Based System Development

managerial problems associated with attempts to integrate a reuse-based software development model with existing software development methodologies such as software development life cycle and prototyping. Technical issues relating to the identification, classification, retrieval, understanding, modification, and integration steps in figure 2 are discussed in more detail in the next section, while the nontechnical issues are covered in the three sections following that.

Software Reuse Technologies

AS MENTIONED EARLIER, REUSE TECHNOLOGIES CAN BE DIVIDED INTO TWO CLASSES based on the nature of the resources being reused: generation technologies and composition technologies. Generation technologies reuse general problem-solving patterns (rather than previously written code) to produce target systems. These are "white-box" technologies—the programmer's task is to recognize the applicable patterns, to fill in the details and supply values for needed parameters. Advanced-language systems, application generators, and transformation systems are examples of generation technologies. Generation technologies are well explained in [8, 52]. We will not discuss them further here.

Composition technologies reuse previously developed and stored software resources such as data-flow diagrams, subroutines, object class hierarchies, and objects. The programmer's task is to recognize the needed "black-box" resources and to integrate them into the overall software product. Software libraries and the software

factory concept outlined above are examples of composition technologies. In this section we discuss reuse libraries and then the composition technologies that can be used to support the reuse-oriented steps of the software development process in figure 2.

Reuse Libraries

Reuse libraries may be implemented in general programming environments such as COBOL or UNIX [50], in conventional database management systems [45], as specialized information retrieval systems [31], object class libraries [61], object repositories [65], or CASE tool repositories [38]. Software libraries often have as few as 100 reusable components, but there are examples of libraries with 1,000 or more components [97]. The physical form of the library affects the classification and retrieval possibilities as discussed below.

Regardless of the physical implementation of the library, there are major decisions to be made concerning the type of software resources to be stored in the library and the application domains to be supported. In most existing software libraries, the resources to be reused are atomic code fragments (subroutines, functions, objects) that are usually unchanged in the course of their reuse [9]. For example, program component libraries such as the SPSSX and IMSL software packages have been very effective in statistical and mathematical applications. Generic software resources such as user interface components and date routines are also relatively easy to reuse. Object component libraries containing such generic resources are being used as the basis for client-server systems development in many companies. Moving in the direction of higher abstraction, code "skeletons" (e.g., for update, report and select, and/or edit programs) that provide a pattern for the solution of a particular class of programming problems but need tailoring to fit a particular problem can provide large productivity gains [53]. "Application templates" (complete system designs developed in a CASE environment) have also been found to be reusable with large gains in efficiency [38]. This supports the idea mentioned earlier that the reuse of system artifacts from all stages of the software life cycle can be highly profitable.

An important tradeoff occurs with regard to the size of objects stored in the repository. The larger the component, the higher the payoff when it is reused. However, large components tend to be more specific, which narrows the range of applications and tends to increase the need for costly modifications [21].

Two recent developments promise to relieve organizations of some of the burden of implementing their own reuse repositories. The first is the development of "object request brokers" that receive requests from remote clients, invoke the appropriate object, and return the results of the request. The objective is to promote sharing, interpretability, and portability. The OMG Group's Common Object Request Broker Architecture (CORBA) specification is an attempt to develop an industry-wide standard for commercial repositories of objects that can interoperate over a distributed environment [65]. The second development is the advent of Java on the World Wide Web. This provides a new and potentially valuable source of reusable software resources making it possible to create distributed object-oriented applications that

function independently of particular operating systems or hardware platforms. In this sense, Java can support the concept of widespread software reusability better than other programming languages.

Identifying and Developing Reusable Resources (Step 1 in Figure 2)

Reusable resources are often acquired in an ad-hoc fashion as a byproduct of new software development or by scavenging old code on an as-needed basis. However, in the spirit of planned reuse, they can also be obtained by prespecifying and developing what is important in a given application area (domain analysis) or by systematically recovering resources from existing software (software reengineering and reverse engineering).

Domain analysis is a process by which information describing an application domain is identified, captured, and organized with the purpose of reusing it in new systems [95]. Domain analysis is related to systems analysis but has a much broader scope and different outputs. The latter include a "domain model" consisting of standard terminology, and reusable artifacts such as data models, generic functions, and architectural hierarchies. Approaches to domain analysis include library analysis [74], object-oriented design [98], and knowledge elicitation [66]. These approaches and industry experiences in domain analysis are discussed in [39, 95]. Domain analysis is a slow, unstructured learning process, and it is difficult to find software developers who have sufficient business knowledge to perform it effectively. However, the payoffs from a successful domain analysis can be substantial. To date, software reuse has been most successful in narrow and well-understood application domains with standardized architectures where domain analysis can be most effective [29]. There is a need to further develop formal approaches to domain analysis together with supporting methods and tools.

In software reengineering, old, poorly written source code is automatically "restructured" (made more maintainable and reusable) without changing its functionality [19]. Reverse engineering goes one step further by transforming existing software code into more abstract and general resources such as cross-reference listings, structure diagrams, and data-flow diagrams [19, 34]. As mentioned above, such abstract resources may be more valuable than concrete resources [62]. Automated reverse engineering tools are summarized in [7]. In a somewhat different approach, Caldiera and Basili [17] developed a quantitative model for identifying reusable software components in existing code. This involves metrics for three reusability attributes—functional usefulness, reuse cost, and quality—that can be used to automate the identification of reusable software components. For example, modules that are invoked relatively often and pass given thresholds on the metrics are identified as potential reusable resources and passed to a human domain expert for further analysis, testing and packaging.

After a reusable resource has been identified, considerable effort may be required to generalize, document, and test it [19]. Generalization of a component usually involves separation of the application and machine environment aspects from the underlying functionality of the component followed by parameterization.

Classifying and Retrieving Reusable Resources (Steps 2 and 4)

The classification and retrieval problem has been widely researched in library science. The problem is to have one person (the classifier) classify documents in such a way that other people can recognize their relevance to the problem at hand and retrieve them. The major classification schemes of library science have all been used in the context of software reuse. The enumerative method (hierarchical classification as in the Dewey Decimal System) and key-word methods have been used successfully in highly or moderately structured domains [45]. To provide freedom in developing ad-hoc categories, a significant amount of research has been done on faceted classification schemes. For example, in [75], each software resource is characterized by six "facets": <function, objects, medium, system type, functional area, setting> with a standard vocabulary for each facet. A similarity function is used to describe how closely each available resource corresponds to a given query. Key-word and faceted classification schemes are advantageous in that they provide indexing terms with well-known semantics. However, they require manual effort in indexing and understanding of the terms by users. Reuse retrieval mechanisms based on full-text or inverted index retrieval provide the possibility of automated approaches to classification, although, in practice, they often require that a standard text "prologue" (author name, date, description, etc.) be stored with each resource [31].

Specialized approaches to the classification and retrieval problem have been developed in the software reuse field. For example, the entity-relationship model has been used to capture the generalization/specialization and part-of relationships that are inherent in software systems [77], and case-based reasoning has been proposed to retrieve resources that are similar but not exact matches for a user's description of the target resource [57]. The discovery and reuse of specification resources such as data-flow diagrams held in CASE repositories by a process of pattern matching through analogy is described in [58]. In a different approach, hypertext interfaces to reuse libraries have been proposed by several researchers [22]. The hypertext paradigm supports nonlinear exploration of a network of associations between information objects and has a natural fit to the problem of maintaining relationships and semantic associations among software resources from all stages of the development life cycle.

The most comprehensive approach to the classification and retrieval problem is to build a specialized information system that records design and structural information about existing software resources [51]. For example, the LaSSIE (Large Software System Information Environment) system uses frame-based knowledge representation and reasoning to simplify the knowledge engineer's task during construction and to provide a knowledge structure and semantic retrieval capabilities for subsequent browsing, navigation, and query reformulation [25].

Finally, it is important to consider criteria for measuring the effectiveness of different classification and retrieval strategies. In a test of retrieval effectiveness using the STAIRS (full-text retrieval) system in a legal application, the average recall ratio (i.e., the proportion of relevant retrieved items to the number of relevant items) was 20 percent, and the average precision ratio (i.e., the proportion of relevant retrieved

items to the number of retrieved items) was 79 percent [10]. Ideally, one would like both recall and precision to equal one. There are a number of factors in the software reuse domain that might lead to efficient retrieval performance. First, the number of relevant resources for a given software retrieval request is generally much lower than in a traditional library application. Second, the domain of software reuse may be more structured, with a larger potential for the use of standardized terms that would aid retrieval. On the other hand, there may be a stronger possibility that there are no relevant items in the software library that can satisfy a given request. This is because of the limited scope of software libraries when compared with the diversity of possible software needs. In an experiment to evaluate the retrieval effectiveness of a free-text indexing scheme for software libraries, the maximum recall ratio was approximately 88 percent, and the precision ratio was between 52 percent and 84 percent [56]. More experiments on the effectiveness of the various classification-retrieval schemes such as those mentioned above are needed to guide further development of reusable software retrieval methodologies.

Understanding Reusable Resources (Step 5)

Reusable software resources are worthless if they cannot be understood and if the likelihood of their successful incorporation into a new software system cannot be assessed. To improve understanding, software resources should be developed for reuse from the beginning, based on the technologies for identification, classification, and retrieval of reusable software resources discussed in the previous section. The greatest benefits from software reuse are likely to be obtained if large and complex resources appear to be relatively simple to the user. This implies the need to provide views of reusable resources that correspond to the mental models of developers [14]. To provide multiple views of software resources that may or may not have been developed explicitly for reuse, Ramamoorthy et al. [77] developed an "information abstractor" that scans code resources and stores information into a database for later querying. Because hypertext systems can integrate information about diverse software resources such as descriptions, diagrams, and explanations of decisions, they may indirectly support a user's mental model [8].

A final task that needs support in this step of the reuse process is assessing the suitability of the resource in the target system and estimating any required conversion costs [19].

Modifying and Integrating Reusable Resources (Steps 6 and 8)

After reusable resources are retrieved and understood, they must be modified and integrated into the target system. Unfortunately, the complex process of integrating reusable resources into the target system is usually left entirely to the software developer. Current technologies for modifying reusable resources focus mainly on parameterized code resources. However, even highly parameterized software modules are shaped by prior implementation decisions and can be difficult to modify for use

in a new context. Object-oriented languages such as C++ attack this problem by using message passing and inheritance as integrating principles. The UNIX pipe mechanism provides a limited form of integration in which one program's outputs are connected to another program's inputs to construct more complex programs [50]. Another promising approach is adopted in the PARIS system, which maintains a library of programs in which some parts remain abstract and undefined [49]. PARIS provides an interactive mechanism to search through the library for a schema that can be reused and supports the refinement and conversion of non-program abstract entities in the retrieved schema to concrete source programs.

Summary of Software Reuse Technologies

A large number of approaches for identifying, classifying, retrieving, understanding, and integrating reusable resources are being actively researched. This research seems to be at a formative stage. There is a need to determine the most effective techniques in each of these phases of the reuse process and to develop an integrated and standardized approach that can be readily understood and adopted by a large community of software developers. Since reuse requires understanding and matching of requirements with stored resources, there is also a need to match the language of classification and retrieval to the language of the requirements specification.

Behavioral Issues

ATTEMPTS TO INTRODUCE A PROGRAM OF SOFTWARE REUSE MAY FAIL more from a lack of management support and software developers' interest than for technical reasons [1, 41]. A detailed analysis of the implications of behavioral and cognitive science theories that are relevant to successful reuse is obviously beyond the scope of this paper. Instead, we will simply assume that workers will do what they want to do and what they can do [83]. What they want to do depends on a host of factors including the social context of work, the drive to satisfy individual needs, intrinsic and extrinsic rewards linked to performance, and educational programs focusing on the benefits of reuse. What they can do in the reuse area is limited by their cognitive abilities and is impacted positively both by education on reuse techniques and by support tools that lighten the cognitive load associated with producing and using reusable resources.

Management Support

The importance of management commitment for the successful implementation of systems projects has been demonstrated by IS implementation research [43]. Similarly, without the support of management and their willingness to make reuse investments, it is very difficult for software reuse to succeed [35]. Software managers will view software reuse differently depending on their position in the organization. Because top-level managers have a long-term strategic viewpoint, it is relatively easy for them to understand the benefits and to support efforts to increase software reuse

[32]. However, middle-level managers may have difficulty supporting software reuse because of a short-term focus on individual projects and performance evaluations that emphasize meeting target completion dates and budgeted costs [1]. They may also fear that higher degrees of reuse will reduce the need for software developers and affect their power in the organization [78]. One approach to obtaining management commitment is to provide managers with convincing evidence of its usefulness through examples of successful software reuse projects, cost-benefit studies, and return-on-investment calculations [35]. Another approach is through education.

Motivational Issues

A number of negative motivational biases inhibit reuse. First, the producers of reusable resources may not wish to share their intellectual contributions with others. On the consumption side, the "not invented here" syndrome describes resistance to the use of reusable software for reasons of ego or because of concerns about quality and reliability of software resources developed by others [97]. Software developers may also fear deskilling and worry about job security as a consequence of a software reuse program [6]. On the plus side, they stand to gain through increased personal productivity and the exposure to others' software development experience and domain knowledge.

The literature in the area of organizational behavior contains many competing theories of motivation and its relationship to task performance (an indication of the difficulty of the area and a warning that simple prescriptions may not work). For example, social information processing theories [83] hold that individuals adapt their attitudes about work to the social context and to the reality of their own past and present actions. Another set of theories emphasizes need satisfaction and recognizes the existence of both extrinsic needs (such as monetary rewards) and intrinsic needs (such as self-actualization) [62]. We now visit each of these issues, social context and incentives, in turn.

Social Context

It is important to develop a climate in which reuse is regarded as a necessary and beneficial practice from both the individual and organizational perspectives. Management commitment is obviously an important component of social context. Other actions that management can take include the development of standard reuse practices, public acknowledgment of progress in reuse, and educational programs. If reuse is successful enough, it may eventually be taught regularly in standard degree programs and become accepted as part of the software engineering "culture," thus reducing the burden of proof for management.

Incentives

One possibility for improving the chance of successful software reuse is to develop a program of nonmonetary and/or monetary rewards [1]. Incentives can create an

encouraging atmosphere for software reuse in the organization, demonstrate management support of software reuse, and symbolize organizational recognition of the importance of reuse [41]. Intrinsic (or psychological) rewards for good reuse performance by individuals or teams are exemplified by praise from superiors, honorable mentions in the company news media, prizes, medals, and so on. Many organizations provide such rewards, and it is generally agreed that they can be effective if not overdone. Provided an appropriate measurement system is in place, extrinsic, monetary incentives can be developed for producers (who develop reusable software resources), consumers (who use reusable software resources in new applications), project managers (who directly supervise producers and consumers of reusable software resources), and contractors (who can be both producers and consumers). Producers may receive both cash bonuses when a software resource is accepted for reuse in the software library and royalties when that resource is reused in new applications. For example, GTE Data Services pays producers a cash bonus of $25 each time a software resource is reused [46]. Consumers can also be rewarded when they reuse software resources in the software library [96]. Budget increases and promotions are good incentives for project managers. The lack of incentives for contractors has been one of the major impediments to successful software reuse in outsourcing situations.

Although most authors support the use of incentive schemes, the evidence from a recent survey shows that few firms have formal incentive or recognition programs and those that have such programs have found that they provide limited benefits [32]. It seems that more research is needed on the impact of monetary and nonmonetary rewards on reuse success.

Cognitive Issues

Even with a favorable climate and the best of intentions, software reuse can be limited because it is difficult from a cognitive standpoint [23, 97]. This applies to both the producers (who must understand the application domain) and consumers (who must understand the reusable resources before they can reuse them.) Human information-processing capabilities and, especially, the capacity limitation of short-term memory are important deterrents to software reuse [25]. The perceived complexity of software can be reduced by chunking or modularization of resources. This argument is directly related to information hiding or abstraction [71] and to the object-oriented approach to reuse. Conceptual differences and the ambiguities of natural language that cause different people to describe the same thing in different ways are at the heart of the classification/retrieval problem in software reuse. Fischer et al. [28] describe two prototype systems based on cognitive principles that help users locate and comprehend software resources. Soloway and Ehrlich [89] carried out an empirical study of the differences between expert programmers and novices. They suggest that expert programmers have at least two types of knowledge: programming plans and rules of programming discourse. When experts develop applications, they try to match pieces of the problem with solution segments with which they are familiar [88]. This implies

that portions of designs are consciously reused when applications are developed. It might be hypothesized that maintaining a database of such previous designs will facilitate performance and learning by software developers at all levels of proficiency.

Standardization of the vocabulary and techniques for reusability can help overcome a number of problems in recognizing and understanding reusable resources. Some progress is being made in this area. An IEEE working group on reuse is developing standards for the reuse of code, documentation, test sets and test environments, knowledge, and skills [59].

Education

To date, few if any degree programs or professional education programs have courses devoted to software reuse. As a result, there is little appreciation on the part of the software industry either of the possible benefits or of the inherent difficulties involved in instituting a program of reuse. Appropriate training improves software reuse and is a necessary part of a reuse program [32]. This is borne out by industry experience in successful reuse programs. The Toshiba Software Factory, which has one of the most comprehensive approaches to reuse, provides extensive and continual training in reuse with specialized courses and personal reviews tied to a plan for individual career advancement [60].

According to Tracz [94], the following topics should be included in a training program for software developers: domain analysis, creating reusable software resources, composing new systems from reusable software resources, quality assurance, documentation, and reuse policies, procedures, and practices. Managers also need to be trained for the purpose of defining their roles, obtaining their commitment, and informing them of reuse technologies. They should also learn how to perform cost-benefit analyses, manage cultural change, and develop reporting systems.

An important issue is the appropriate forum for reuse education. Reuse is not likely to become a major component of university degree programs until it has been proved in practice. In the meantime, reuse education will probably be performed by in-house or external consultants and will be based on company-specific rather than industry standard practice. However, the U.S. Department of Defense has developed both general and ADA-specific reuse education programs [2] and commercial object repositories such as CORBA [65] have associated educational seminars.

Summary of Behavioral Issues

The introduction of a software reuse program presents managerial as well as technical challenges. Research is needed on the determinants of management commitment, the effectiveness of the various forms of software reuse incentive programs, and the content and modes of delivery of reuse training programs. We also need to understand how to resolve potential conflicts among people involved in software reuse such as managers, software developers, project managers, and contractors. On the technical side, there is a need for tools to reduce the complexity of expressing requirements and

to assist users in finding reusable resources and understanding the software systems they are attempting to build. Finally, we need to understand educational needs and to develop appropriate training programs.

Organizational Issues

A NUMBER OF ORGANIZATIONAL FACTORS MAKE IT HARD TO IMPLEMENT successful software reuse programs. If widespread reuse is to be achieved in practice, there may be a need to change the infrastructure of the organization to admit new organizational roles with responsibilities for developing standards, maintaining the reuse library, and reporting results.

Organizing for Reuse

There are a number of different ways to organize for reuse, ranging from doing nothing (relying on the ad-hoc efforts of individual developers) to a full-scale reorganization involving complete standardization of all phases of the software development process. Since reuse requires that different organizational units reuse common resources, coordination, communication, and education are needed for even a small-scale reuse program. Most authors suggest that successful reuse requires the creation of distinct roles such as corporate sponsor, reuse librarian, reuse coordinator, and reuse task force member [48]. Reuse task groups can be organized for each project or each product class or application domain. Depending on the size of the software organization and the significance of the reuse effort, these roles can be informal or formal and may represent either part-time or full-time activities for those concerned. Centralized reuse support groups have been created in several organizations with a strong commitment to reuse. Examples include a central support group at AT&T [82] and a Corporate Reuse Council at IBM [93]. To provide coordination, centralized groups usually consist of representatives from each major project or application domain. The duties of a reuse support group include identifying and managing reusable software resources, promoting their use, developing standards, managing the software library, educating others, and providing help-desk support [64, 80].

The most ambitious applications of reuse techniques probably occur when organizations adopt an industrial metaphor to reorganize their software development processes. As mentioned earlier, Japanese companies such as Toshiba and Hitachi have created "software factories" that achieve high levels of software reuse using concepts borrowed from engineering and manufacturing industries [24]. The idea is to gain economies of scope by sharing standard software resources and processes across families of related software products. Software is developed using a series of simple repeatable processes that are continuously improved by collecting and analyzing detailed metrics for productivity, reuse, and quality [60]. It seems that the factory concept can transcend cultural boundaries as it has been adapted successfully by the European consortium Eureka [26] and by Hewlett-Packard [34], among others [27]. These organizational innovations are reminiscent of the organizational changes that

occurred when manufacturing moved from a craft industry to mass production at the beginning of the century. It remains to be seen if they will be broadly successful in the rapidly evolving software industry.

Reuse Support Environments

In addition to creating specialized groups and new organizational structures to support reuse, a number of organizations have developed support environments consisting of reuse repositories and search and retrievals tools such as those discussed earlier. An effective reuse support environment helps overcome the problems associated with traditional organizational design: poor alignment and communication between producers and consumers of reusable software resources, the absence of standards for metric analysis and quality assurance of reusable software resources, and the lack of an environment to institutionalize software reuse [6, 68].

Reuse Capability Maturity Models

As described above, reuse practices in companies vary from ad-hoc, occasional reuse by individual programmers to planned and carefully executed programs of reuse involving new organizational entities and investments in the development of reusable resources and a supporting environment. It is important for organizations to understand their progress in the area of reuse relative to other companies and industry best practice. In the software area, the "Capability Maturity Model (CMM)" developed by SEI [72] describes the evolution of an organization's development processes from a chaotic ad-hoc state to a state of maturity in which industry-wide best practices are the norm. Similar maturity assessment models have been developed for reuse: Some models are add-ons to the SEI CMM, while others have reuse as the major theme [48]. To date, there has been little experience of CMM in the reuse area. However, if such models can be validated, practitioners and researchers will have a valuable tool to determine the overall state of reuse in individual organizations and in the software industry as a whole.

Measuring the Value of a Reuse Program

While a reuse maturity model can measure the progress of an organization in developing reuse processes, it is also important to measure the ongoing costs and benefits of a reuse program. Most organizations that recognize the value of reuse collect data on levels of reuse achieved and overall software productivity. However, few have developed reporting systems that encourage and assess the performance of both the producers and consumers of reusable resources. An assessment model developed by Poulin et al. [73] serves to illustrate the issues. Maintaining the notation used earlier where possible, their model is as follows:

1. For each project and organizational unit, the effectiveness of the reuse effort

is measured by two parameters, the reuse rate, R and the dollar benefits due to cost avoidance, RCA:

Reuse cost avoidance, $RCA = DCA + SCA$;

Development cost avoidance, $DCA = RSI * (1–b) *$ (Cost per LOC);

Service cost avoidance, $SCA = RSI *$ (Error rate) * (Cost per error),

where:

RSI = Reused source instructions (unchanged from the reuse library).

2. For each project and organizational unit, its effectiveness as both a user and producer of reusable parts is measured by the ratio:

$$\text{Reuse value added, } RVA = \frac{SSI + RSI + SIRBO}{SSI}$$

where:

SSI = Shipped source instructions minus the total new noncomment lines in the finished product (not including reused source instructions);

$SIRBO$ = software instructions used by others = (SSI per part) * N.

Here N is the number of organizations that reuse the part. Note that $RVA > 1$. $RVA = 1$ corresponds to the case of no reuse. An organization can perform well on this metric if it is effective either as a consumer ($RSI > 0$) or a producer ($SIRBO > 0$) or both. This tends to level the playing field across software teams who have different opportunities for developing new reusable resources.

3. For each project and organizational unit, the increased cost of producing reusable components is given by:

$$ADC = (E - 1) * SIRBO * (\text{Cost per } LOC).$$

4. For each project, the overall dollar benefit from reuse (called ROI in [73]) is given by:

$$ROI = RCA + RCA_O - ADC,$$

where:

RCA_O = the reuse cost avoidance obtained by other projects' use of the $SIRBO$ produced by this project (see [73] for formula).

We have explained this model at length because it illustrates some of the difficulties in developing an effective set of metrics and an accounting system for reuse. The model requires the periodic computation of four control numbers (R, RCA, RVA, and ADC) for each organizational unit (programming team) and project with suitable aggregations for reporting at higher levels in the organization. The four control factors are computed from nine metrics, three of which (cost per LOC, error rate, and cost per error) should normally be available for each project and organizational unit in a mature

software development organization. The total noncomment source instructions (*SSI* + *RSI*) can also be maintained in the normal course of affairs using a simple code counting program; separating out the *RSI* obtained from the software library should present no problem. The other six metrics (*SSI*, *RSI*, *SIRBO*, *E*, *B*, *N*) are additional accounting requirements imposed by the need to monitor the reuse program. In [73], the reported industry values for *E* (1.5) and *b* (0.2) are suggested as proxies for the actual values attained by each programming team; obviously, this is an approximation that avoids considerable additional administrative work.

The definitions given above highlight some subtle issues. First, *RSI* included in the software product as a macro or subroutine is counted only once (no matter how many times the module is called), and a call to a subroutine counts as just one *SSI*. Second, *N* is defined as the number of *organizations* (software teams) that use the software component independent of the number of times the component is used by either the producing team or by other teams. This avoids inflated reuse values for components as it is equivalent to assuming that teams will always remember components that they have used at least once and will avoid the cost of redeveloping a similar component in a different context. However, it is conservative in that the use of a reusable component by an organization in different projects is not captured. A final set of issues concerns how these variables are used. Should management set absolute target values or percent improvement goals for the four measures? How is the information used to encourage good performance? Is it the basis for incentive schemes and/or linked to salary adjustments and career advancement?

The above is probably the simplest model that could effectively measure the ongoing effectiveness of a software reuse program and the contributions of each project and organizational unit as consumers and producers of reusable software. Many variations on this model are possible, as are more complex schemes such as [13, 48]. This area represents a particularly important and potentially fertile area for research as the lack of standard reuse metrics is one of the major impediments to successful software reuse [81].

Summary of Organizational Issues

Organizational structures to support reuse are evolving in many organizations with little understanding of the issues that affect different choices and no clear patterns to guide management choice. Much could be learned from a comparative study of organizational adaptations for reuse across different organizations facing different software tasks. For example, it would be interesting to see if the paradigm shift represented by the software factory concept can be successfully applied on a broad scale. Another set of interesting research issues concern the efficacy of reuse maturity models and the design of management control and reporting mechanisms in the reuse arena.

Legal and Contractual Issues

THE BASIC IDEA BEHIND SOFTWARE REUSE INVOLVES THE SHARING of intellectual products. However, increasing the reuse potential of software resources makes it easier

for unauthorized individuals to use them. In a commercial situation, preparing software resources for reuse can facilitate access by competitors and decrease the competitive advantage of the original developers. The current trend toward outsourcing software development services exacerbates these problems. Unless proper arrangements and precautions are taken, a software reuse program can become seriously embroiled in a host of security and legal issues [97]. This section briefly discusses two issues: protecting reusable resources from illegal use and developing legal arrangements to facilitate sharing of resources among parties to a contract.

Software Protection

Developing reusable resources requires considerable effort and results in software assets that are easy to appropriate for illegal as well as legal purposes. Among other issues, the producers of reusable resources need to be concerned with the possibility of illegal reuse of their software through the process of reverse engineering. A company that develops reusable resources either in a contract with a third party or for resale in the market place can protect its investment using one or more of three forms of intellectual property protection: trade secrets, copyrights, and patents. Trade secret protection is available for information that is not commonly known and is kept relatively secret [99]. Trade secret rights are limited to prevention of copying and cannot be used to prevent another's independent development of the protected information. Therefore, it may be permissible to reverse-engineer lawfully obtained software to discover trade secrets [84].

Since 1980, computer programs have been explicitly covered by the federal copyright law. Copyright protects original works of authorship against unauthorized copying but does not protect ideas such as algorithms, methods, or concepts [100]. Two caveats limit the protection provided by copyrights. First, under the "fair use doctrine," the public can have the right to copy a protected work for private noncommercial purposes [85]. Second, while legal opinions differ, it may be legal under certain conditions to reverse-engineer someone else's software. This is particularly the case if the reverse engineering is performed via "black-box" testing of inputs and outputs [84].

Patents provide stronger protection than copyrights or trade secrets because patents may be violated even if there is no copying [42]. Software-related inventions are judged for patentability in the same way as non-software-related inventions by the Supreme Court [99]. Patent claims cannot be made for pure mathematical algorithms, the operation of a computer, a human performing the same steps, or a method of doing business. Although it is time-consuming and relatively expensive to obtain patents, patents are probably a more appropriate means of protection for widely marketed software products such as reusable object repositories than trade secrets or copyrights.

Contractual Issues

The importance of contractual issues related to the use and ownership of reusable software resources is beginning to be recognized by organizations (such as the United

States Department of Defense) that are heavily dependent on contracting for software systems development, and by organizations (contractors) that develop and provide software systems to these organizations. The current trend toward outsourcing software development makes this a pertinent issue for business as well. When multiple countries are involved in software projects (e.g., the European Space Agency which has eleven member countries), contractual issues are particularly complicated.

The objective of the contracting organization is to develop integrated systems using common reusable parts for improved maintainability. At the same time, it wants to force the contractor to use existing reusable resources (to reduce costs) and to make the reusable resources developed under the contract available for use in future contracts. However, if the contracting organization acquires all rights to the software systems delivered by the contractor, there is little motivation for that contractor to develop reusable resources. As more software resources are developed for reuse, it is important to preserve contractors' rights across a wide variety of software projects, and to motivate contractors by providing financial rewards for practicing software reuse [37]. For example, in the European Space Agency, two concepts ("background" and "foreground") are used to protect contractor rights. Software resources that are generated under a given contract are called foreground; software resources that are generated outside the contract, but are used for the contract, are called background. In general, the contractor is the owner of the development results (foreground), while the contracting organization has a nonexclusive license to use the results. Since the contractor makes use of background resources, the contracting organization has to acquire the right to use them.

Another issue is the reuse of customer-owned software resources by contractors. Depending on contractual requirements (i.e., optional or mandatory use of reusable resources), responsibility for the functioning of software resources will be different. If the contractor wants to reuse certain customer-owned software resources, the contractor will be responsible for their correct functioning in the target software product. The customer, of course, should provide appropriate documentation for proper evaluation of the software resources. On the other hand, if the customer mandates the reuse of certain customer-owned software resources, the customer will be responsible for their correct functioning [92].

Summary of Legal and Contractual Issues

This section has highlighted the need for research in two areas. First, research is needed on how to reconcile the conflicting objectives of easier access to reusable resources and protection from unauthorized use. Second, there is a need to develop contractual forms that encourage reuse by ensuring that producers and consumers of reusable resources are treated equitably.

Conclusions and Research Directions

SOFTWARE REUSE IS A VERY COMPLEX AND STILL poorly understood topic. While it is not a panacea, it is probably the most promising way to improve software development

Table 3. Summary of Findings

Software reuse issue	Impact of issue on the success or failure of software reuse	What we know from prior research	What we need to know/develop
Definition and scope	Different definitions and scopes of software reuse can lead to confusion and lack of focus in research and make communication with practitioners difficult.	Existing reuse frameworks are based either on type of software artifact or type of reuse technology. Managerial and economic issues are important. Researchers emphasize the reuse of a broad range of software resources while practitioners focus on code reuse.	Can we develop a broadly accepted framework and terminology to integrate and communicate research results in the software reuse area? Which areas of technical, organizational, and behavioral research will have the greatest payoff? How can we develop a broad range of reusable resources, and especially abstract resources?
Benefit and cost	Large investments in software reuse programs need to be justified. Management needs to measure and control the impact of a program of software reuse.	Several organizations have successfully monitored reuse but the practice is not widespread. Economic models of code reuse support the potential for large gains and there have been a number of reports of these gains being realized in practice.	What metrics are needed to measure the costs and benefits of reuse? How can we develop metrics for abstract resources? How can such metrics be integrated into the management of software development? Can we develop a comprehensive long-term investment model that will be accepted by management?
Software reuse process	A formal reuse-based software development methodology should increase the chance of reuse success.	Traditional software development methodologies do not explicitly support software reuse. Companies with successful reuse programs have developed formal methods and processes but there are many different approaches.	What are the implications of a formal reuse-based software development methodology that emphasizes activities for both producing and consuming reusable resources? How can reuse-based development methodologies be integrated into existing development methodologies?

Software reuse technologies	The lack of tools and techniques to produce, document, and consume reusable resources is a barrier to successful software reuse.	Software reuse has been most successful in narrow and well-understood application domains. While there is ongoing research into various aspects of reuse support such as the identification, classification, retrieval, and integration of reusable software, no techniques or approaches have emerged as dominant. Methods for formally representing information requirements and specifications are poorly developed, especially for abstract resources which are generally informal in nature.	What mixture of management approaches, methods, and tools can best support reuse activities? How can we reuse the products of former development activities? Can the logic of legacy systems be reused? How can we represent the full range of products from the software development process in such a way that they can be successfully reused? What tools and mechanisms can help the integration of heterogeneous resources into the target software? How can we build a support environment for reuse? Can CASE tools be extended to better support reuse?
Behavioral issues	Attempts to introduce a software reuse program may fail because of managerial and human issues such as management commitment, lack of understanding, absence of incentives, and cognitive overload.	Management commitment is a necessary ingredient for a successful reuse program. Middle-level managers may not support software reuse because of a short-term viewpoint or for political reasons. Software developers do not always have the discipline and cooperative spirit needed to successfully apply reuse principles. Training programs are effective in promoting reuse. Financial and recognition-based incentive systems have been used by several firms.	How can the value of software reuse programs be established and communicated to managers? What do managers and software developers need to understand about reuse? How can training courses best be designed and delivered? How can we introduce software reuse techniques into university programs? How effective are incentive programs in encouraging reuse by individual software developers? How should they be designed? What standards are needed to improve communication and reduce the cognitive complexity of reuse?

Table 3. Continued

Organizational issues	Organizational factors can make it hard to implement successful software reuse.	There must be changes in the infrastructure of the organization for widespread software reuse to occur. Software reuse support groups have been successful in some organizations. Highly formal and mechanical approaches to software development have worked in Japanese "software factories."	How can we best organize to implement a reuse program? What is the role of a reuse support group? How can potential conflicts between reuse support groups and software developers be avoided? What is the tradeoff between formal and informal software reuse processes? Can software factory ideas succeed in different organizations and cultures?
Legal and contractual issues	Unless proper arrangements and precautions are taken, a software reuse program can become seriously embroiled in a host of security and legal issues.	Existing copyright and patent protections are of limited use in preventing unethical use of software resources. Increasing the reuse potential of software resources can decrease the competitive advantage of the original developers.	How can we reconcile the conflicting objectives of easier access to reusable software resources and protection from unauthorized use? What contractual forms can best promote reuse across organizations?

productivity and quality [9, 97]. Though a considerable amount of reuse research has been published over the last twenty years, this has focused, for the most part, on a small subset of reuse issues. This paper has surveyed recent software reuse research using a framework that includes a broad range of technical and nontechnical factors that can have an impact on the success or failure of a software reuse program. Table 3 summarizes the findings from this survey by showing the impact of each critical reuse issue and what we know and need to know about it. Major research opportunities exist in all of the areas of software reusability research in the table.

In our opinion, two major conclusions can be drawn from this survey. First, a reuse program is not likely to succeed unless careful attention is paid to a broad range of nontechnical economic, behavioral, organizational, and legal issues. Thus, the "library paradigm" of software reuse in which a collection of reusable resources is made available to users without further provisions to make it successful is unlikely to work. Second, a more comprehensive approach to the development of reuse support tools is required. We envision a reuse support system (RSS) [51] that helps document and elucidate existing application systems so that the ideas and design decisions involved in their creation can be reused either in the context of maintenance or when building new systems. In the latter case, the reuse support system should encourage the use of standard data definitions and software design approaches both throughout the organization and between organizations. The reuse support system should provide an environment to help users understand unfamiliar software resources and decide whether to use them or not. The LaSSIE system [25] and Draco [69] represent current approaches to building such an "information system about an information system."

Acknowledgments: We wish to thank the Editor-in-Chief and the reviewers for their excellent comments and great patience as this paper went through the review process. Referee number 1 in particular made many helpful suggestions.

NOTES

1. For comparison, the average programmer productivity in the United States is reported to be between 100 to 500 LOC per month of new and reused code [44].
2. For comparison, the average failure rate under "normal practices" can be as high as 20 to 60 fixes per 1,000 lines of code [20].

REFERENCES

1. Apte, U.C.; Sankar, M. T.; and Turner, J. Reusability-based strategy for development of information systems: implementation experience of a bank. *MIS Quarterly, 14*, 4 (December 1990), 421–433.
2. Army Reuse Center. **http://arc_www.belvoir.army.mil/educate.html**.
3. Balda, D., and Gustafson, D. Cost estimation models for the reuse and prototype software development life cycles. *ACM SIGSOFT Software Engineering Notes, 15*, 3 (July 1990), 42–50.
4. Banker, R.D.; Kauffman, R.J.; Wright, C.; and Zweig, D. Automating output size and software reuse metrics in an object-based case environment. *IEEE Transactions on Software Engineering, 20*, 3 (March 1994), 169–187.

5. Barnes, B., and Bollinger, T. Making reuse cost-effective. *IEEE Software, 8*, 1 (January 1991), 13–24.

6. Biffl, S., and Grechenig, T. Degrees of consciousness for reuse of software in practice: maintainability, balance, standardization. *Proceedings of the 17th Annual International Computer Software & Applications Conference*, 1993, pp. 107–114.

7. Biggerstaff, T.J. Design recovery for maintenance and reuse. *Computer, 22*, 7 (July 1989), 36–49.

8. Biggerstaff, T.J., and Perlis, A.J. *Software Reusability*. Reading, MA: Addison-Wesley, 1989.

9. Biggerstaff, T.J., and Richter, C. Reusability framework, assessment, and directions. *Software Reusability Vol. I Concepts and Models*. Reading, MA: Addison-Wesley, 1989.

10. Blair, D.C., and Maron, M.E. An evaluation of retrieval effectiveness for a full-text document-retrieval system. *Communications of the ACM, 28*, 3 (March 1985), 289–299.

11. Boehm, B.W. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

12. Boldyreff, C. Reuse, software concepts, descriptive methods, and the practitioner project. *ACM SIGSOFT Software Engineering Notes, 14*, 2 (April 1989), 25–31.

13. Bollinger, T.B., and Pfleeger, S.L. The economics of reuse: issues and alternatives. *Proceedings of the Eighth Annual National Conference on Ada Technology*, 1990, pp. 436–447.

14. Bott, M.F., and Wallis, P.J.L. Ada and software re-use. *Software Engineering Journal, 3*, 5 (1988), 177–183.

15. Buck-Emeden, R., and Galimow, J. *SAP R/3 System: A Client/Server Technology*. Reading, MA: Addison-Wesley, 1996.

16. Bullard, C.K.; Guindi, D.S.; Ligon, W.B.; McCraken, W.M.; and Rugaber, S. Verification and validation of reusable Ada components. In P.A. Leslie, R.O. Chester, and M.F. Theofanos (eds.) *Guidelines Document for Ada Reuse and Metrics*. Oak Ridge, TN: Martin Marietta Energy Systems, Inc., 1989.

17. Caldiera, G., and Basili, V.R. Identifying and qualifying reusable software components. *IEEE Computer, 24*, 2 (February 1991), 61–70.

18. Cheng, J. Improving the software reusability in object-oriented programming. *ACM SIGSOFT Software Engineering Notes, 18*, 4 (October 1993), 70–74.

19. Chikofsky, E.J., and Cross, J.H. Reverse engineering and design recovery: a taxonomy. *IEEE Software, 7*, 1 (January 1990) 13–17.

20. Cobb, R.H., and Mills, H.D. Engineering software under statistical quality control. *IEEE Software, 7*, 6 (November 1990) 44–53.

21. Coome, T.N.; Comer, J.R.; and Rodjak, D.J. Developing reusable software for military systems—why it is needed and why it isn't working. *ACM SIGSOFT Software Engineering Notes, 15*, 3 (July 1990), 33–38.

22. Creech, M.; Freeze, D.F.; and Griss, M.L. Using hypertext in selecting reusable software components. *Proceedings of Hypertext '91*, 1991, pp. 25–38.

23. Curtis, B. Cognitive issues in reusing software artifacts. *Software Reusability Vol. II Applications and Experience*. Reading, MA: Addison-Wesley, 1989.

24. Cusumano, M.A. *Japan's Software Factories*. New York: Oxford University Press, 1991.

25. Devanbu, P.; Brachman, R.; Selfridge, P; and Ballard, B. LaSSIE: a knowledge-based software information system. *Communications of the ACM, 34*, 5 (May 1991), 34–49.

26. Dabin, M. Software reuse and CASE tools. *Proceedings of the 15th Annual International Computer Software and Applications Conference*, 1991, pp. 2–3.

27. Fafchamps, D. Organizational factors and reuse. *IEEE Software, 11*, 5 (September 1994), 31–41.

28. Fischer, G.; Henninger, S.; and Redmiles, D. Cognitive tools for locating and comprehending software objects for reuse. *Proceedings of the 13th International Conference on Software Engineering*, 1991, pp. 318–328.

29. Frakes, W.B.; Biggerstaff, T.J.; Prieto-Diaz, R.; Matsumura, K.; and Schaefer, W. Software reuse: is it delivering? *Proceedings of the 13th International Conference on Software Engineering*, May 1991, pp. 52–59.

30. Frakes, W.B., and Fox, C.J. Sixteen questions about software reuse. *Communications of the ACM, 38*, 6 (June 1995), 75–87.

31. Frakes, W.B., and Nejmeh, B. Software reuse through information retrieval. *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences*, 1987, pp. 530–535.

32. Freeman, P. Reusable software engineering: concepts and research directions. *ITT Proceedings of the Workshop on Reusability in Programming*, 1983, pp. 129–137.

33. Gaffney, J.E., Jr., and Durek, T.A. Software reuse—key to enhanced productivity: some quantitative models. *Information Software Technology, 31*, 5 (June 1989), 258–267.

34. Gall, H., and Klosch, R. Reuse engineering: software construction from reusable components. *Proceedings of the 16th Annual International Computer Software and Applications Conference*, 1992, pp. 79–86.

35. Griss, M.L. Software reuse: from library to factory. *IBM Systems Journal, 32*, 4, (1993), 548–566.

36. Griss, M.L.; Adams, S.; Baetjer, H. Jr.; Cox, B; and Goldberg, A. The economics of software reuse (Panel). *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1991, pp. 264–270.

37. Griss, M.L.; Favaro, J.; and Walton, P. Managerial and organizational issues—starting and running a software reuse program. In W. Schaefer, R. Prieto-Diaz, and M. Matsumoto (eds.), *Software Reusability*. New York: Ellis Horwood, 1994, pp. 51–78.

38. Hofman, J.D., and Rockart, J.F. Application templates: faster, better and cheaper systems. *Sloan Management Review, 36*, 2 (1992) 41–48.

39. Hooper, J.W., and Chester, R.O. *Software Reuse*. New York: Plenum, 1991.

40. *IMSL (International Mathematics and Scientific Library)*. IMSL Inc., 1991.

41. Isoda, S. Experience report on software reuse project: its structure, activities, and statistical results. *Proceedings of the 14th Annual International Conference on Software Engineering*, 1992, pp. 320–326.

42. Jakes, J.M., and Yoches, E.R. Basic principles of patent protection for computer software. *Communications of the ACM, 32*, 8 (August 1989), 922–924.

43. Jarvenpaa, S.L., and Ives, B. Executive involvement and participation in the management of information technology. *MIS Quarterly, 15*, 2 (June 1991), 205–227.

44. Jones, T.C. Reusability in programming: a survey of the state of the art. *IEEE Transaction on Software Engineering, 10*, 5 (September 1984), 488–493.

45. Jones, G., and Prieto-Diaz, R. Building and managing software libraries. *Proceedings of IEEE COMPSAC*. 1988, pp. 228–236.

46. Joyce, E.J. Reusable software: passage to productivity? *Datamation* (September 15, 1988), 97–102.

47. Karakostas, V. Requirements for CASE tools in early software reuse. *ACM SIGSOFT Software Engineering Notes, 14*, 2 (April 1989), 39–41.

48. Karlsson, E., ed. *Software Reuse: A Holistic Approach*. New York: John Wiley, 1995.

49. Katz, S.; Richter, C.H.; and The, K. PARIS: a system for reusing partially interpreted schemas. *Proceedings of IEEE 9th International Conference on Software Engineering*, 1987, pp. 377–385.

50. Kernighan, B.W. The Unix system and software reusability *IEEE Transaction on Software Engineering, 10*, 5 (September 1984), 513–518.

51. Kim, Y. RSS: an approach to widespread software reusability. Ph.D. dissertation, New York University, 1992.

52. Krueger, C.W. Software reuse. *ACM Computing Surveys, 24*, 2 (June 1992), 131–183.

53. Lanergan, R.G., and Grasso, C.A. Software engineering with reusable designs and code in software reusability. *Software Reusability*, vol. 2, *Applications and Experience*. Reading, MA: Addison-Wesley, 1989.

54. Lim, W.C. Effects of reuse on quality, productivity, and economics. *IEEE Software, 11*, 5 (September 1994), 23–30.

55. Love, T. The economics of reuse. *Proceedings of IEEE COMPCON*, 1988, pp. 238-241.

56. Maarek, Y.S.; Berry, D.M.; and Kaiser, G.E. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering, 17*, 8 (August 1991), 800–813.

57. Mackellar, B.K., and Maryanski, F. A knowledge base for code reuse by similarity. *Proceedings of IEEE COMPSAC*, 1989, pp. 634–641.

58. Maiden, N., and Sutcliffe, A. Exploiting reusable specification through analogy. *Communications of the ACM, 35*, 4 (April 1992), 55–64.

59. Marovac, N. *IEEE Working Group: Development of a Standard for Reuse of Code.* IEEE Technical Council on Software Engineering, 1992.

60. Matsumoto, Y. Some experiences in promoting reusable software: presentation in higher abstract levels. In *Software Reusability*, vol. 2. Reading, MA: Addison-Wesley, ACM Press, 1989.

61. Micallef, J. Encapsulation, reusability and extensibility in object-oriented programming. *Journal of Object Oriented Programming* (April–May 1988), 12–34.

62. Mitchell, T.B. Expectancy models of job satisfaction, occupational preference and effort. *Psychological Bulletin, 81*, 12 (1974), 1053–1107.

63. Miyoshi, T., and Azuma, M. An empirical study of evaluating software development environment quality. *IEEE Transactions on Software Engineering, 19*, 5 (May 1993), 425–532.

64. Moad, J. Cultural barriers slow reusability. *Datamation* (November 15, 1989), 87–92.

65. Mowbray, T.J., and Zahavi, R. *The Essential CORBA: Systems Integration Using Distributed Objects.* New York: John Wiley, 1996.

66. Motta, E.; Rajan, T.; and Eisenstadt, M. Knowledge acquisition as a process of model refinement. *Knowledge Acquisition, 2* (1990), 21–49.

67. Myers, W. We want to write less code, asserts symposium keynoter. *IEEE Computer, 23*, 7 (July 1990), 117–118.

68. Navarro, J.J. Characteristics of a flexible software factory: organization design applied to software reuse. *Proceedings of the 38th IEEE Computer Society International Conference*, 1993, pp. 265–267.

69. Neighbors, J.M. The Draco approach to constructing software from reusable components. *IEEE Transaction on Software Engineering, 10*, 5 (September 1984), 564–574.

70. Nierstrasz, O.; Gibbs, S.; and Tsichitris, D. Component-oriented software development. *Communications of the ACM, 35*, 9 (September 1992), 160–165.

71. Parnas, D.L.; Clements, P.C.; and Weiss, D.M. Enhancing reusability with information hiding. In T.J. Biggerstaff and A.J. Perlis (eds.), *Software Reusability*, vol. 1, *Concepts and Models.* Reading, MA: Addison-Wesley, 1989, pp. 141–157.

72. Paulk, M.C.; Curtis, B.; and Chrissis, M.B. *Capability Maturity Model for Software.* Pittsburgh, PA: Software Engineering Institute, CMU/SEI–91-TR–24, 1991.

73. Poulin, J.S.; Caruso, J.M.; and Hancock, D.R. The business case for software reuse. *IBM Systems Journal, 32*, 4 (1993), 567–594.

74. Prieto-Diaz, R. Domain analysis methodology. *Proceedings of the Workshop on Domain Modeling, ICSE–13*, 1991, pp. 138–140.

75. Prieto-Diaz, R., and Freeman, P. Classifying software for reusability. *IEEE Software, 4*, 1 (January 1987), 6–16.

76. Pyster, A., and Barnes, B. The software productivity consortium reuse program. *Proceedings of IEEE COMPCON*, 1988, pp. 242–247.

77. Ramamoorthy, C.V.; Garg, V.; and Prakash, A. Support for reusability in Genesis. *IEEE Transactions on Software Engineering, 14*, 8 (August 1988), 1145–1154.

78. Rauch-Hindin, W.B. Reusable software. *Electronic Design, 31*, 3 (February 1983), 176–193.

79. Redwine, S.T., Jr., and Riddle, W.E. Software reuse processes. *Proceedings of ACM Software Process Workshop*, 1989, pp. 133–135.

80. Reifer, D.J. *Managing Software Reuse: A Case Study Approach.* New York: John Wiley, 1993.

81. Rubin, K. Reuse in software engineering: an object-oriented perspective. *Proceedings of IEEE COMPCON*, 1990, pp. 340–346.

82. Ryan, D. *RAPID/NM, Reusable Architectures for Transaction Processing and Network Management Applications.* Murray Hill, NJ: AT&T, 1990.

83. Salancik, G.R., and Pfeffer, J. A social information processing approach to job attitudes and task design. *Administrative Science Quarterly, 23* (1978), 224–253.

84. Samuelson, P. Reverse-engineering someone else's software: is it legal? *IEEE Software, 7*, 1 (January 1990), 90–96.

85. Samuelson, P. Interface specifications, compatibility, and intellectual property law.

Communications of the ACM, 33, 2 (February 1990), 111–114.

86. Schaefer, W.; Prieto-Diaz, R.; and Matsumoto, M. *Software Reusability*. New York: Ellis Horwood, 1994.

87. Selby, R.W. Quantitative studies of software reuse. In T.J. Biggerstaff and A.J. Perlis (eds.), *Software Reusability*, vol. 2, *Applications and Experience*. Reading, MA: Addison-Wesley, 1989, pp. 213–233.

88. Soloway, E., and Ehrlich, K. What do programmers reuse? theory and experiment. *Proceedings of ITT Workshop on Reusability in Programming*, 1983.

89. Soloway, E., and Ehrlich, K. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering 10*, 5 (September 1984), 595–609.

90. Sprague, R.H., Jr., and McNurlin B.C., eds. *Information Systems Management in Practice*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

91. Strassman, P. *The Squandered Computer: Evaluating the Business Alignment of Information Technologies*. New Canaan, CT: Information Economics Press, 1997.

92. Syms, T., and Christine, L.B. Software reuse: customer vs. contractor point-counterpoint. *Proceedings of Ada Euro Conference*, 1990.

93. Tirso, J.R. Championing the cause: making reuse stick. *Proceedings of the 5th Annual Workshop on Software Reuse*, 1992, pp. 1–6.

94. Tracz, W.J. Where does reuse start? *ACM SIGSOFT Software Engineering Notes, 15*, 2 (April 1990), 42–46.

95. Tracz, W. J. Domain analysis working group report—first international workshop on software reusability. *ACM SIGSOFT Software Engineering Notes, 17*, 3 (July 1992), 27–34.

96. Tracz, W.J. Second international workshop on software reusability—IWSR2 summary. *ACM SIGSOFT Software Engineering Notes, 18*, 3 (July 1993), A73–A77.

97. Tracz, W.J. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Reading, MA: Addison-Wesley, 1995.

98. Wirfs-Brock, R., and Wilkerson, B. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

99. Yoches, E. Legal protection for computer software. *Communications of the ACM, 32*, 2 (February 1989), 169–171.

100. Yoches, E., and Levine, A.J. Basic principles of copyright protection for computer software. *Communications of the ACM, 32*, 5 (May 1989), 544–545.