J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*
Copyright © 1999 John Wiley & Sons, Inc.

SOFTWARE LIBRARIES

A *software library* is a managed collection of software assets where assets can be stored, retrieved, and browsed. A *software asset* is a usable representation of some software development experience.

While the most typical form of software asset is a piece of source or executable code, this definition allows for much wider characterizations. Any object that embodies some form of software development experience qualifies as an asset; this includes system requirements, component specifications, design descriptions, source-code components, executable-code components, test data, process control data, cost estimation data, and management plans.

Software engineers should, as a matter of routine, have the capability to harness the combined wisdom of others. There are several uses for this capability: to become familiar with a programming language or a programming language style, to look for common patterns of usage, to get acquainted with an application domain, or to reuse library assets (rather than creating them from scratch). However, without a procedure to systematically record the construction of software and associated assets, it becomes difficult to extract any knowledge that is potentially helpful in subsequent engineering activity.

The fundamental role of software libraries is then to store and communicate experience and outcomes—in essence, to serve as a vehicle for the exchange of ideas between groups of software engineers. However, even with a systematic regime of capturing useful ideas in a software library, successful transfer of ideas between software engineers requires significant organizational effort.

Fundamentally, it is the process of communicating the information that demands the effort. Communication of ideas requires overheads: the language in which ideas are expressed must be understood by all participants in the exchange, and the ideas themselves should be clear and unambiguous, especially when pertaining to software systems. It is therefore important that software libraries record ideas and outcomes generated during software developments in such a way as to enable other software engineers to understand them. In turn this can significantly ease the process of software engineering, by providing a source of quality software assets.

Since software libraries serve as an archive for software development experience, each software library can also be regarded as a software information system—that is, a repository of information to be used in the study of software systems. Consequently, there are two ways to avail oneself of a software library: by retrieving assets that satisfy a predefined, preformulated query; or by navigating through the library to acquaint oneself with its content.

Software libraries have long been considered as the key component of any software reuse initiative, where their role is to serve as a repository of assets that may be reused. It is more appropriate to consider, perhaps, that while they play an important role in software reuse, software libraries are not generally key success factors; typically, they play a secondary role in the success of a software reuse initiative to such considerations as team organizations, software processes, managerial structures, or human factors.

Characteristic Software Library Features

A software library can be characterized by a number of features, discussed below. While these attributes are not strictly orthogonal, they do give a sense of the complex issues that arise when one defines a software library. These issues are discussed in detail in Mili et al. (1).

(1) *Nature of the Asset* The most important feature of a software library is, of course, the nature of assets that are stored therein. The most typical asset is code (be it source code or executable code), but other kinds of assets are also possible: specifications, designs, test data, documentation, and so on. Some library methods are restrictive, in the sense that they work for a single kind of asset, whereas others may work for a wide range of assets.

(2) *Scope of the Library* Another crucial feature of a software library is its scope: whether the library is expected to be used within a single project, within an organization, or on a larger scale. In order to use a software library effectively, a reuser must share some common knowledge with the maintainer of the library (e.g., pertaining to the interpretation of terminology, the representation of assets, the form and meaning of results, etc.) and with other users. If this common knowledge is general (so that any reuser is likely to have it), then one expects the library to have a large scope; the more specialized this knowledge, the smaller the scope.

(3) *Query Representation* A software library can be characterized by the form that queries submitted to the library must take. Among possible options we may mention: a formal functional specification, a signature specification, a behavioral sample, a natural-language query, or a set of keywords.

(4) *Asset Representation* The representation of assets is an important feature of a library, not only because it dictates what form user queries take, but also because it determines how retrieval is performed. In a perfectly transparent library the representation of an asset is irrelevant to the user, but no library is perfectly transparent. The representation of an asset is often called the *surrogate* for that asset. Among the possible values of this attribute, we mention: formal specifications, signature specifications, the set of keywords, the source text, the executable code, and requirements documentation.

(5) *Storage Structure* The most common logical storage structure in software libraries is to have no structure at all: software assets are stored side by side with no ordering between them. While in traditional database systems entries are ordered by their identifying keys, it is difficult to define a general key that can be used to order software assets in a meaningful way. There are some exceptions: some libraries that are based on formal specifications order assets by the refinement ordering between their specifications; and artificial-intelligence-based software libraries define semantic links between assets.

(6) *Navigation Scheme* This attribute is correlated to *storage structure*, because the storage structure determines to a large extent the navigation scheme of the method. In flat storage structures, the only possible pattern of navigation is brute-force exhaustive search of all the entries. Whenever the assets are arranged on some nontrivial structure, this structure can be used to help orient the search towards those assets that are most likely to satisfy the query—and steer it clear from those that are known to be irrelevant or are thought unlikely to be relevant.

(7) *Retrieval Goal* To fix our ideas, we discuss this feature in the context when the library assets are programs. In principle, the goal of a retrieval operation is to find one or several programs that are correct with respect to a given query. If this retrieval operation fails to turn up candidate programs, one may want to perform another retrieval operation, with the lesser ambition of finding programs that approximate the query, with the expectation that we must modify the (selected) retrieved programs. Depending on whether we are interested in generative modification or compositional modification, the goal of the retrieval operation changes considerably: under generative modification, we are interested in programs whose design can

be adapted to solve the query; under compositional modification, we are interested in programs whose components can be combined to solve the query.

(8) *Relevance Criterion* The relevance criterion states under what condition a library asset is considered to be relevant for the submitted query with respect to the predefined retrieval goal. In reference to the discussion above about generative and compositional modification, observe that under generative modification, assets are deemed relevant (or not) on the basis of their structure, whereas under compositional modification assets are deemed relevant (or not) on the basis of their function.

(9) *Matching Criterion* The matching criterion is the condition that we choose to check between the submitted query and a candidate library asset to decide whether the asset is to be retrieved. Ideally the matching criterion should be equivalent to the relevance criterion, but it is not always so: if the asset's surrogate is too abstract, and/or if the relevance criterion is too intractable, these two criteria may differ significantly. For example, if the relevance criterion is correctness of library assets with respect to the submitted query, and if library assets are represented by (arbitrarily abstract) functional specifications, then it is tempting to let the matching criterion be that the asset's surrogate (its specification) must be a refinement of the query. This matching criterion is sufficient to ensure relevance (since it logically implies the relevance criterion), but is not necessary: it is possible that the asset does indeed satisfy all the requirements of some query, yet, because it is too abstract, the asset's surrogate (which does not record all the features of the asset) fails to refine the query; this is an instance where an asset satisfies the relevance criterion (since it is correct with respect to the query) but fails to satisfy the matching criterion (since its surrogate does not refine the query).

To summarize this section, we present in Table 1 a listing of all the attributes discussed above, along with a tentative (not necessarily exhaustive) indication of the values that each attribute may take.

Library Organizations

Using the attributes listed in Table 1, we have divided the existing library organization methods into six classes. Each class corresponds to a distinct pattern of attribute values and includes all the library organizations that fit the pattern; where we discuss a class, we present the pattern of attributes that characterize it. We have identified six classes of storage/retrieval methods, which we present below, in the order of increasing technological sophistication.

(1) *Information Retrieval Methods* These are methods that depend on a textual analysis of software assets. It is important to acknowledge that the storage and retrieval of software assets is nothing but a specialized instance of information storage and retrieval. Hence it is important to discuss these methods, and possibly highlight their shortcomings: If traditional information retrieval methods were adequate in dealing with software assets, there would be little incentive to investigate other methods.

(2) *Descriptive Methods* These are methods that depend on a textual description of software assets. While information retrieval methods represent assets by some form of text, descriptive methods rely on an abstract surrogate of the asset, typically a set of keywords or a set of facet definitions. Also, while information retrieval methods select assets by attempting to *understand* them (in the sense of natural-language processing), descriptive methods merely attempt to *characterize* candidate assets. This has a profound influence on the design as well as the performance of retrieval algorithms.

(3) *Operational Semantics Methods* These are methods that depend on the operational semantics of software assets. They can be applied to executable code, and proceed by matching candidate assets against a user query on the basis of the candidates' behavior in response to sample inputs. This technique is called *behavior*

Table 1. Attributes of a Software Library

Attribute	Characterization
Nature of asset	Source code, executable code, requirements specification, design description, test data, documentation, proof
Scope of library	Within a project, across a program, across a product line, across multiple product lines, worldwide
Query representation	Functional specification, signature specification, keyword list, design pattern, behavioral sample
Asset representation	Functional specification, signature specification, source code, executable code, requirements documentation, keywords
Storage structure	Flat structure, hypertext links, refinement ordering, ordering by genericity
Navigation scheme	Exhaustive linear scan, navigating hypertext links, navigating refinement relations, navigating with catalogs
Retrieval goal	Correctness, functional proximity, structural proximity.
Relevance criterion	Correctness, signature matching, minimizing functional distance, minimizing structural distance
Matching criterion	Correctness formula, signature identity, signature refinement, equality/subsumption of keywords, natural-language analysis, pattern recognition

sampling. Behavior sampling methods constitute an elaboration on information retrieval methods, in the sense that they exploit a unique feature of software assets, namely their executability.

(4) *Denotational Semantics Methods* These are methods that depend on the denotational semantic definition of software assets. Unlike operational methods, they can also be applied to nonexecutable assets (such as specifications). These methods proceed by checking a semantic relation between the user query and a surrogate of the candidate asset. The surrogate of the software asset can be a complete functional description, a partial functional description, or a signature of the asset.

(5) *Topological Methods* The main discriminating feature of topological methods is their goal, which is to identify library assets that minimize some measure of distance to the user query. This feature, in turn, has an effect on the relevance criterion, and hence on the matching criterion. Whether an asset is relevant cannot and need not be decided by considering the query and the candidate asset alone, since the outcome depends on a comparison with other assets.

(6) *Structural Methods* The main discriminating feature of structural methods is the nature of the software asset they are dealing with: Typically, they do not retrieve executable code, but rather program patterns, which are subsequently instantiated to fit the user's needs. This feature, in turn, has a profound effect on the representation of queries and assets, as well as on the relevance criterion, which deals with the structure of assets and queries rather than their function.

It is fair to say that most software libraries that are in use nowadays are instances of the first two or three classes discussed above.

Managing Software Libraries

The task of managing software libraries is viewed from the perspective of an organization considering the adoption of software library technology. There are both economic and technical concerns.

Library Management: Economic Aspects. Like many activities of software engineering, the management of software libraries is driven by economic considerations. Economic considerations form the basis of all decision making that pertains to library management, including: building a library (an institutional decision); storing an asset (a domain engineering decision); searching the library (an application engineering decision). We review these three decisions, and discuss the economic considerations that they raise, without going into the details of cost modeling.

(1) *Institutional Decision* The creation of a software library can be seen as an investment decision, where investment costs include the cost of the library infrastructure and the up-front domain engineering that is required to populate the library. Periodic costs include the manpower required to operate the library, weighted against the benefits that the organization gains from exploiting the library assets.
(2) *Domain Engineering Decision* Producing an asset and storing it in the library is also an investmentlike decision. Investment costs include the cost of producing or acquiring the asset and storing it. Periodic costs include the impact that an extra entry in the library has on the library performance, as well as (more importantly) the potential of this asset to be a distraction in subsequent retrievals, due to poor retrieval precision (because of imperfect retrieval precision, the asset may be retrieved without being relevant—hence causing undue distraction to the software engineer). Periodic benefits include, of course, the quality and productivity gains that are achieved by reusing the asset, prorated by the frequency with which the asset is retrieved and used.
(3) *Application Engineering Decision* Resolving to use a software library for the purpose of a software project carries the potential of productivity and quality gains, but is not without risk: if there is a mismatch between the project's application domain and the library's, or if the retrieval precision is very poor, then using the library may cause a loss of productivity by distracting the software engineer with irrelevant assets. There is also the cost of understanding and adapting library assets if and when they are found to be relevant.

Library Management: Technical Aspects. The task of managing a collection of software assets, in a technical sense, is similar to managing a collection of bibliographic entries in a library catalog. Of paramount concern in both these management tasks is the skill and art of organizing the managed assets for ready access. A library (software or other) is of little use unless assets can be retrieved when and only when they are relevant to a user.

To support application engineering decisions, consideration of techniques for organizing, cataloging, and classifying traditional library holdings are relevant when considering the management of software assets in a software library. It is the process of cataloging that dictates the organization of traditional libraries; software libraries may also use catalogs as the basis for organizing software assets. Specifically, large-scope software libraries have catalogs, each catalog containing asset surrogates as entries. The catalogs can be used as a library navigation scheme (see Table 1). Quality criteria for traditional library catalogs are then also directly transferable to the domain of software libraries: these criteria for software library catalogs are discussed below.

(1) *Flexible and Current* Entries in the catalog should be current (this requires flexibility to update a catalog) and should be complete in the sense that (as far as is possible) no relevant entries are absent.
(2) *Accessible* The catalog should be constructed so that all entries can be quickly and easily found. The catalog should avoid unnecessary impediments to visibility of entries, thereby minimizing the effort to retrieve, assess and select software assets.
(3) *Economic* The effort invested to manage a catalog must be far less than the effort to construct the software assets.
(4) *Detailed* Entries should contain macro- and micro-level information about the assets described. For instance, rather than an entry describing a computer program as an asset, an entry could also permit access to descriptions of definitions used within the program.

The purpose of maintaining cataloged asset collections for large-scope software libraries is to bring related assets together in a helpful sequence from the general to the specific, with the ultimate aim of leading the software engineer to relevant assets.

Designing Software Libraries

The task of designing a software library involves five steps, which are introduced below.

(1) *Define the Assets* Given a particular organizational setting, not all kinds of development experience will be included as assets in a library. A software library designer must specify the kinds of development experience that qualify as assets for the purposes of the software library being built.
(2) *Model the Assets* To design a library of assets, one must choose a representation for each kind of asset, and store assets in a database according to the chosen representation. A representation of assets that lacks detail may preclude certain asset manipulation operations, while an asset represented in too much detail unduly complicates asset access operations.
(3) *Define Relationships between Assets* Having chosen an asset representation, a software library designer must define and organize interasset relationships. Relationships between software assets can be used to implement browsing and other access mechanisms, and can also help detect incompleteness or inconsistency amongst software library asset holdings. These relationships are particularly useful if the library contains more than one kind of asset.
(4) *Decide Asset Insertion, Removal, and Access Policies* Having defined the assets and relationships of interest, a software library designer must decide upon policies for the major software library operations: insertion, removal, and access of assets.
(5) *Decide Asset Update Policies* As the managed asset collection evolves over time, the definition and control of interasset relationships and constraints in the presence of asset updates need to be addressed. Policies to manage relationships in the presence of change must preserve the ability to present descriptions of assets and the relationships among them at any stage of the library's history.

Within the framework of these five steps, a number of further design decisions must be made. These decisions involve how to specify integrity constraints over library holdings, and how to determine the relevant factors when defining asset insertion, removal, access, and update policies. These design decisions are discussed in the following sections.

Integrity Constraint Design. As an information system, a software library has integrity constraints that must be maintained by any library operation. These constraints form the basis for insertion, removal, access, and update policies.

Constraints for insertion are concerned with controlling entry of submitted assets into the library; constraints for removal are concerned with controlling exit of assets from the library; constraints for access are concerned with controlling querying of assets in the library; constraints for updates are concerned with controlling manipulation of assets in the library. These constraints determine the extent to which each library activity is controlled. The library designer is obligated to define integrity constraints on the software library and its assets to enable the detection, assessment, and repair of constraint violations.

The primary goal of integrity constraint design is to maintain a consistent library state. A number of other factors affect the choice of constraints:

(1) *Semantic Correctness* For assets where the notions are meaningful, software library assets must be well formed and well typed; that is to say, semantically correct. Integrity constraints can be used to enforce this as policy.
(2) *Asset Comprehension* The act of legislating consistent "well-styled" forms of expression using integrity constraints can often aid asset comprehension. Even simple kinds of stylistic constraints (such as standard, consistent naming conventions and standard document structures) assist asset comprehension.
(3) *Development Standards* A software library is often used in a software development process. The tasks and activities in such a process may require coding and other development standards for the purposes of traceability, uniformity, and integrity of the process. Such requirements can be enforced using integrity constraints.
(4) *Library Integrity* The integrity of the software library needs to be ensured. For example, attempts to insert assets already held in the library should be refused. These conditions can be enforced using integrity constraints.

Asset Insertion Policies. Asset insertion is a filtering process. Through the use of constraints, a software library insertion policy for assets is produced. These constraints act as a filter of the library by refusing insertion of those assets that violate the constraints.

However, it is not sufficient to merely detect constraint violations and to reject violating assets. Software engineers expect feedback that states a precise reason for rejection, and that reveals ways of improving the rejected asset. Therefore assessment and repair of constraint violations is also a part of the insertion process.

Having created an asset, a software engineer may submit it to a library for insertion. Submission of a new asset may represent a serious modification to the library, affecting relationships with other engineered assets. Upon submission, there should be a check to determine whether the submitted asset (or a sufficiently similar asset) is already stored in the library; if so, the submitted asset will be rejected. It is only when an integrity constraints can be maintained that an asset submission is successful.

To summarize, the process of insertion involves the following steps:

(1) *Submission* An asset is created and submitted to the software library. This asset is the candidate for insertion.
(2) *Library Integrity Check* An attempt to retrieve the candidate asset from the library using some retrieval mechanism is then made.
(3) *Refusal* If a similar asset is found to be within the library, the candidate asset is refused admission to the library. The reason for refusal is presented to the software engineer, who may then consider using the similar asset so retrieved in place of the candidate asset.

(4) *Asset Integrity Checks* A collection of integrity constraints is used to certify submitted assets.
(5) *Rejection* If integrity constraints fail, the asset insertion request is rejected, and the user is given information on how to amend the asset.
(6) *Acceptance* Having passed the necessary checks, the asset is stored into the library.

The aim of the insertion process is to safeguard the quality and utility of the software library, by refusing assets that violate constraints. The motivations and sophistication of the means used to specify constraints determines the extent to which software library quality is maintained.

Asset Removal Policies. Removal is a change-propagation process. Removal of an arbitrary asset from a library may violate many constraints; the violations may in turn trigger other asset removals or modifications to reestablish library consistency, as defined by the constraints. A removal policy is formed by specifying the conditions under which each kind of asset can be removed, and through specification of the effects of removal upon other library assets. By specifying stringent conditions under which removal may occur, the propagation of change is controlled, and constraints are maintained. Removal of an existing library asset represents a serious modification to the library. Over time, repeated need for asset removal may reveal inadequacies with insertion policies; if such assets need to be removed, why were they admitted in the first place?

In some circumstances, it may be appropriate to defer or avoid removal, seeking more expedient means to remedy a problem. For example, the asset to be removed may be superseded. In this case, a new asset designed to replace the functionality of the removed asset exists. The superseded asset may remain in the library, but be marked as an older version or as obsolete, to protect applications outside the library that depend upon the superseded asset.

Under other circumstances, it may be inappropriate to avoid removal of an asset. The asset may have been obsolete or unused for a period of time, have been inserted mistakenly, or have become anachronistic. In these cases a removal policy is required. A removal policy comprises a kind of asset to remove, a set of constraints that must hold for the removal to be permitted, and a specify effect of the removal which must precisely govern and limit the propagation of change to related assets as a result of removal.

To summarize, the process of removal involves the following steps:

(1) *Removal Decision* The motivation for removal of the asset is examined. Removal cannot be justified when other means provide better protection of library integrity and client applications.
(2) *Policy Satisfaction* Does the asset meet the required constraints for removal from the library?
(3) *Removal* The asset is removed from the library and the asset collection is changed as specified by the removal policy.

Asset Access Policies. The problem of software library access can be stated as: how does one locate in a software library those assets that are of relevance to the information needs of the library user?

Software engineers aim to identify those assets that are most helpful to their current needs. However, it is possible that such assets will not exist in the software library. In this case, the goal of the software engineer is to search for assets that can, in a straightforward manner, be modified and composed to fulfill their information needs.

The objective of software library access mechanisms is therefore not to strive for near-automation of the search process, but rather to permit software engineers control of the strategic search processes used during browsing and retrieval.

Browsing versus Retrieval. Given a body of software assets, there are two ways in which one may want to avail oneself of these assets: *browsing* and *retrieval*. Whether we choose one or the other of these two options has a significant influence on how the assets ought to be organized. It is possible to characterize the difference between browsing and retrieval in the following terms:

(1) *Retrieval* The software engineer has a predefined set of relevance criteria, and is seeking to retrieve all the assets that satisfy these criteria; these criteria may pertain to the function, the performance, the representation, the structure, or any other property of the assets. Retrieval is consistent with top-down design, and occurs after a system design; it attempts to identify assets that will be used to fill requirements defined by the proposed design. Generally speaking, retrieval requires a library organization where assets are placed in such a way that comparison of an asset with a query allows us not only to make a determination on the asset at hand, but also on a large (as large as possible) set of other assets as well. For the purposes of retrieval, software libraries are best structured by means of ordering relations; ideally, the ordering provides that if some asset is found to be nonrelevant with respect to some query, then so are all the assets that are lower than it in the selected ordering.

(2) *Browsing* The software engineer has a predefined concept of some application domain or some implementation platform, and is seeking to retrieve all the assets that pertain to the application domain at hand, or can operate on the platform at hand. Depending on whether the selection criterion is relevance to an application domain or compatibility with an operating platform, we want to address a library that is organized around a vertical product line or a horizontal product line. Browsing takes place prior to the design of a system, and proceeds to identify assets that may be used in the design; it is consistent with the bottom-up design discipline, and aims to orient the design in such a way as to take the best advantage of available assets. Generally speaking, browsing requires a classification system similar to the Library of Congress classification scheme: all the assets that are relevant to a particular application domain (vertical reuse) or a particular operating platform (horizontal reuse) should be placed together. For the purposes of browsing, software libraries are best organized by equivalence relations; this must be qualified by the premise that, unlike equivalence classes, distinct families of assets may have elements in common.

Browsing, and the bottom-up design discipline that it supports, appear to be more adequate for software reuse than for retrieval: it is more natural to evolve a design around existing assets than to evolve it from the top down and then try to find assets that fit it. Yet, paradoxically, most of the research on software reuse libraries has focused on retrieval rather than browsing.

Asset Update Policies. Policies to manage asset updates have dual concerns: first, a notion of software library consistency with respect to its asset holdings must be defined, and second, techniques for controlling changes to the asset holdings must be decided upon. The goal is to minimize updates while maintaining consistency. This is also the goal of software configuration management research.

Generally, software configuration management concerns the control of all software assets throughout the system life cycle, to preserve the definitions of asset versions and the relationships among them. The effect of software configuration management techniques upon the processes of insertion and removal is now discussed.

Two major activities of software configuration management are consistency checking and change control. Their application to software libraries is now discussed.

Consistency checking supports change control by ensuring that managed constraints between assets are correctly enforced. In practice, such consistency may be assured using tools which may require user interaction. For example, the monitoring of constraints may be undertaken by a human, who visually scans submitted assets.

The process of asset insertion outlined the need for defining a set of integrity constraints over assets that together define consistency for a given state of a software library.

Change control is concerned with the precise definition and control of the effects of changes made to an interrelated collection of software assets. The assumption is that the collection of assets is in a consistent initial state, and the desire is to determine a consistent final state that not only adequately implements the desired change but does so with minimal incremental update of the collection of software assets.

The process of asset removal outlines the need for defining a set of removal policies which together specify the change control mechanisms for a software library. Consistent initial and final states are enforced via the integrity constraints defined for asset insertion, and the effect of change is precisely captured through the specification of constraints in removal policies.

Related Work

Frakes and Gandel (2) survey some methods of software storage and retrieval, using a classification into three families: *library- and information-science indexing methods*, *knowledge-based methods*, and *hypertext methods*. Frakes and Pole (3) revisit the classification and propose a set of assessment criteria that include, in addition to precision and recall, *effectiveness*, *overlaps*, and *searching time*.

Krueger (4) and Mili, Mili, and Mili (5) present surveys of software reuse in general, in which they discuss the storage and retrieval of software assets for the purpose of reuse. Because software libraries are not the focus of their surveys, they are treated at a general level, and take account of the state of the art at their time of publication.

Mili, Mili, and Mittermeir (1) present a survey on software reuse libraries, in which they survey the field, identify software-library characteristics, discuss the different techniques for library organization, and review the literature on software library access methods.

The study of software libraries has mostly taken place under the umbrella of software reuse (see Software Reusability). Updates to library assets over time require consideration of software asset maintenance concerns (see SOFTWARE MAINTENANCE). The process of collection of software assets in a library can be seen as a software packaging activity. Asset retrieval and browsing are method for accessing library assets; more information is available at The Component Retrieval and Reuse web site, located at http://www.cs.tu-bs.de/softech/crrw/.

BIBLIOGRAPHY

1. A. Mili R. Mili R. T. Mittermeir A survey of software reuse libraries, *Ann. Softw. Eng.*, **5**: 349–414, 1998.
2. W. B. Frakes P. B. Gandel Representing reusable software, *Inf. Softw. Technol.*, **32**(10): 653–664, 1990.
3. W. B. Frakes T. P. Pole An empirical study of representation methods for reusable software components, *IEEE Trans. Softw. Eng.*, **20**(8): 617–630, 1994.
4. C. W. Krueger Software reuse. *ACM Comput. Surv.*, **24**(2): 131–183, 1992.
5. H. Mili F. Mili A. Mili Reusing software: Issues and research directions, *IEEE Trans. Softw. Eng.*, **21**(6): 528–561, 1995.

S. ATKINSON
A. MILI
West Virginia University