reprinted from

COMPUTER

Identifying and Qualifying Reusable Software Components

Gianluigi Caldiera and Victor R. Basili
University of Maryland

Effective reuse of knowledge, processes, and products from previous software developments can increase productivity and quality in software projects by an order of magnitude. In fact, software production using reusable components will probably be crucial to the software industry's evolution to higher levels of maturity.

Software reuse is not new. McIlroy[1] proposed using modular software units in 1969, and reuse has been behind many software developments. However, the method has never acquired real momentum in industrial environments and software projects, despite its informal presence there.

The first problem we encounter in reusing software arises from the nature of the object to be reused. The concept is simple — use the same object more than once. But with software it is difficult to define what an object is apart from its context.[2] We have programs, parts of programs, specifications, requirements, architectures, test cases, and plans, all related to each other. The reuse of each software object implies the concurrent reuse of the objects associated with it, and informal information traveling with the objects. Thus, we must reuse more than code. Software objects and their relationships incorporate a large amount of experience from past development. We need to reuse this experience in the production of new software. The experience makes it possible to reuse software objects.[3]

A second major problem in code reuse is the lack of a set of reusable components, despite the large amount of software that already exists in the portfolios of many software producers. Reuse efficiency and cost effectiveness require a large catalog of available reusable objects.

Software metrics provide a way to automate the extraction of reusable software components from existing systems, reducing the amount of code that experts must analyze.

In this article, we outline a way to reuse development experience along with the software objects it produces. Then, we focus on a problem in the development of a catalog of reusable components: how to analyze existing components and identify ones suitable for reuse. After they are identified, the parts could be extracted, packaged in a way appropriate for reuse, and stored in a component repository. This catalog of heterogeneous objects would have to be designed for efficient retrieval of individual components, but that topic is beyond our scope.

Our model for reusing software components splits the traditional life-cycle models into two parts: one part, the project, delivers software systems, while the other part, the factory, supplies reusable software objects to the project. The factory's primary concerns are the extraction and packaging of reusable components, but it must, of course, work with a detailed knowledge of the application domain from which a component is extracted.

Our approach to identification and qualification of reusable software is based on software models and metrics. Because software metrics take into account the large volume of source code that must be analyzed to find reusable parts, they provide a way to automate the first steps of the analysis. Besides, models and metrics permit feedback and improvement to make the extraction process fit a variety of environments.

The extracted candidates are analyzed more carefully in the context of the semantics of the source application in a process we call "qualification."

In this article, we describe some case studies to validate our experimental approach. They deal with only the identification phase and use a very simple model of a reusable code component, but our results show that automated techniques can reduce the amount of code that a domain expert needs to evaluate to identify reusable parts.

0018-9162/91/0200-0061$01.00 © 1991 IEEE

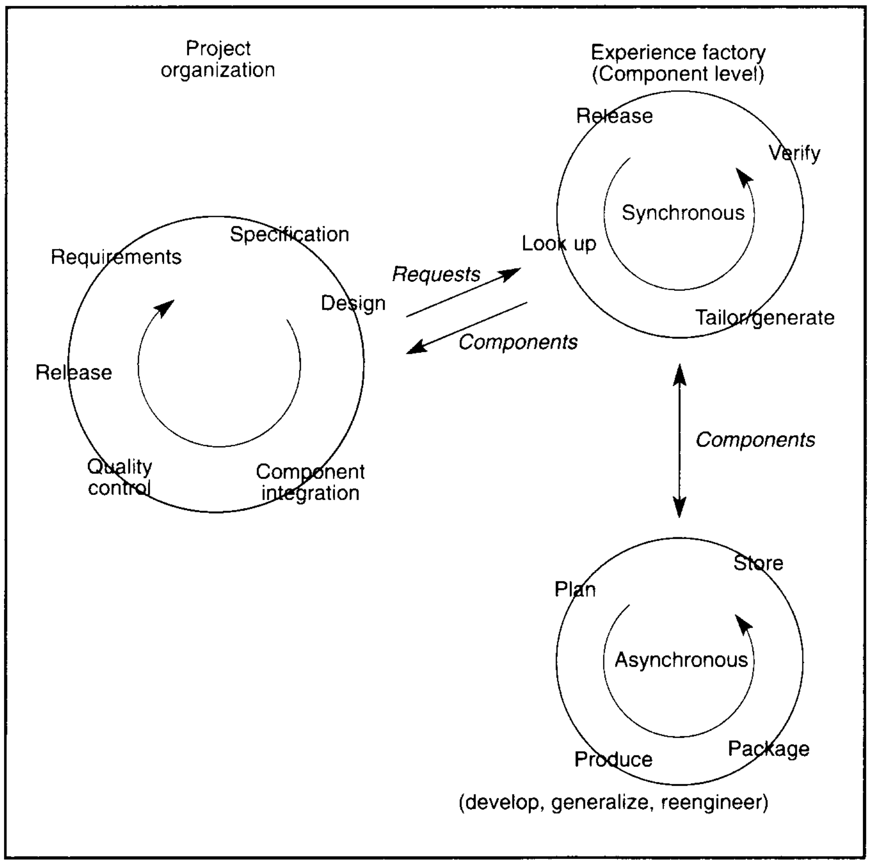


Figure 1. The reuse process model.

Reuse framework and organization

In many software engineering projects, reuse is as common as in everyday life: It is an informal sharing of techniques and products among people working on the same or similar projects. Transforming informal reuse concepts into a technology of reuse would provide the basis for the future software factory, improving quality and increasing productivity, as well as making production more manageable. To achieve higher levels of reuse, we must recognize the experience appropriate for reuse, package experience in a readily available way, and formally integrate reuse into software development.

Currently, all reuse occurs in the project development, where reuse is difficult because a project's focus is system delivery. Packaging reusable experience is at best a secondary concern. Besides, project personnel cannot recognize the pieces of experience appropriate for other projects.

Existing process models, which tend to be rigidly deterministic, are not defined to take advantage of reuse, much less to create reusable experience. To create packaged experience and then reuse it, multiple process models are necessary.

Figure 1 shows an organizational framework that separates project- specific activities from the reuse-packaging activities, with process models that support each activity.[4] The framework defines two separate organizations: a project organization and an experience factory.

The project organization develops the product, taking advantage of all forms of packaged experience from prior and current developments. In turn, the project offers its own experiences to be packaged for other projects. The experience factory recognizes potentially reusable experience and packages it so it is easy for the project organization to use.

Within the experience factory, an organization we call the component factory develops and packages software components. It supplies code components to the project upon demand, and creates and maintains a repository of components for future use. As a subdivision of the experience factory, the experience that the component factory manipulates is programming and application experience as embodied in programs and their documentation. Because the experience factory gathers all kinds of experience from the project, the component factory understands the project context and can deliver components that fit.

The project organization performs activities specific to implementation of the system to which it is dedicated. It analyzes the requirements and produces the specifications and the high-level system design. Its process models are like those used by today's software engineering projects (for instance, it may use the waterfall model or iterative enhancement model). Software engineers generate specifications from requirements and design a system to satisfy those requirements. However, when the engineers have identified the system components, usually after the so-called preliminary design, they request components from the component factory and integrate them into the programs and the system they have designed. The project organization engineers may also request a list of components that satisfy a given specification. Then, from several design options, they can choose the one for which more reusable components are already available.

After component integration, the project organization process model continues as usual with product quality control (system test, reliability analysis) and release.

The component factory's process model is twofold:[3] it satisfies requests for components coming from the project organization, but it also prepares itself for answering those requests. This mix of synchronous and asynchronous activities is typical of the process model of the experience factory in general.[4]

Synchronous activity. When the component factory receives a request from the project organization, it searches its catalog of components to find a software component that satisfies that request with or without tailoring. Two kinds of tailoring can be applied to a software component: instantiation and modification. To an extent, the component's designer has anticipated instantiation by associating with the component some parameters to make it suit different contexts. A generic unit in Ada is an example of such a parametric component and of the instantiation process. Modification is an unanticipated tailoring process in which statements are changed, added, or deleted to adapt the component to a request.

If no component that approximates the

request can be found in the catalog of the available components or if the necessary modification is too expensive, the component factory develops the requested component from scratch or generates it from more elementary components. After verification, the component is released to the project organization that requested it.

Asynchronous activity. The component factory's ability to efficiently answer requests from the project organization is critical for the successful application of the reuse technology. Therefore, the factory's catalog must contain enough components to reduce the chances that the factory will have to develop a component from scratch. Moreover, looking up components must be easy. This is why the component factory's process model has an asynchronous part.

To produce some software components without specific requests from the project organization, the component factory develops a component production plan — it extracts reusable components from existing systems or generalizes components previously produced on request from the project organization. The Booch components[5] are an example of a component production plan: The most common data structures and the main operations on them have been implemented as Ada packages.

A component factory can develop an application-oriented component production plan by analyzing an application domain to identify the most common functions. Then it can implement these functions into reusable components to be used by the developers. Or the factory can generalize a preexisting component into a new one by adding more functionality or parameterizing it.

To ensure that the generated components are well packaged and easily retrieved, a process called component qualification provides components with functional specifications and test cases, and classifies them according to a component taxonomy.[6] Software components are then stored in a repository.

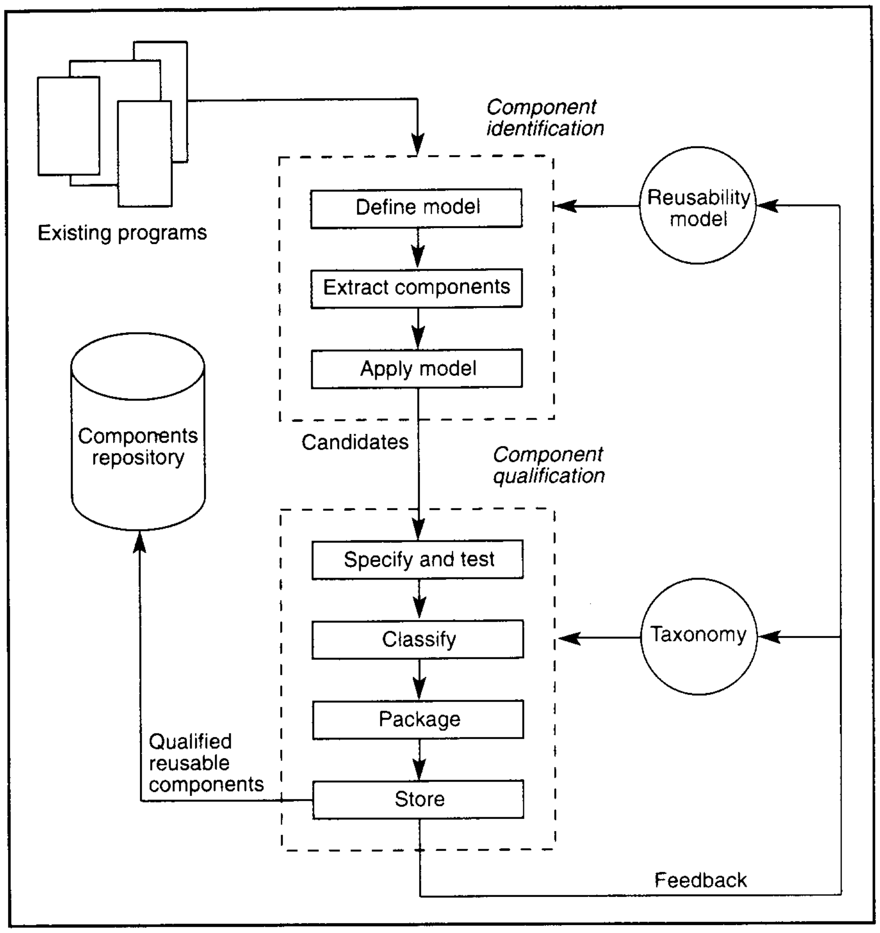


Figure 2. Component extraction.

Extracting components

In the short term, developing reusable components is generally more expensive than developing specialized code, because of the overhead of maintaining the component factory. A rich and well-organized catalog of reusable components is the key to a successful component factory and a long-term economic gain. But at first such a catalog will not be available to an organization, unless it can reuse code that it developed in the past without reuse in mind.

Mature application domains, where most of the functions that need to be used already exist in some form in earlier systems, should provide enough components for code reuse. In such cases, the earlier systems were probably designed and implemented by reusing code informally. For example, Lanergan and Grasso found rates of reuse of about 60 percent in business applications.[7]

To package such code for reuse, the component factory analyzes existing programs in the two phases shown in Figure 2. First, it chooses some candidates and packages them for possible independent use. Next, an engineer with knowledge of the application domain where the component was developed analyzes each component to determine the service it can provide. Then, components are stored in the repository with all information that has been obtained about them.

The first phase can be fully automated. The necessary human intervention in the second phase is the main reason for splitting the process in two steps, instead of searching through existing programs looking for "useful" components first. The first phase reduces the amount of expensive human analysis needed in the second phase by limiting analysis to components that really look worth considering.

In the component identification phase, program units are automatically extracted, made independent, and measured according to observable properties related to their potential for reuse. There has been much discussion about these properties. According to Prieto-Diaz and Freeman,[6] a software component is reusable if the effort required to reuse it is remarkably smaller than the effort required to implement a component with the same functions. Thus, we need a quantitative measure of the distance of the component from its potential reuse. In the section below on component identification, we give details about a family of such measures that we call the reusability attributes model.

The identification phase consists of three steps:

(1) *Definition (or refinement) of the reusability attributes model.* Using our current understanding of the characteristics of a potentially reusable component in our environment, we define a set of automatable measures that capture these characteristics and an acceptable range of values for these metrics. We verify the metrics and their value ranges using the outcomes in the next steps and continually modify them until we have a reusability attributes model that maximizes our chances of selecting candidate components for reuse.

(2) *Extraction of components.* We extract modular units from existing systems, and complete them so they have all the external references needed to reuse them independently (for example, to compile them). By "modular unit" we mean a syntactic unit such as a C function, an Ada subprogram or block, or a Fortran subroutine.

(3) *Application of the model.* The current reusability attributes model is applied to the extracted, completed components. Components whose measurements are within the model's range of acceptable values become candidate reusable components to be analyzed by the domain expert in the qualification phase.

During the component qualification phase, a domain expert analyzes the candidate reusable components to understand and record each component's meaning while evaluating its potential for reuse in future systems. The expert also repackages the component by associating with it a reuse specification,[8] a significant set of test cases, a set of attributes based on a reuse classification schema, and a set of procedures for reusing the component.

The reuse classification schema, called a taxonomy, is very important for storing and retrieving reusable components efficiently. The definition and the domain of the attributes that implement the taxonomy can be improved each time an expert performs component qualification and analyzes the problems encountered.

The qualification phase consists of six steps:

(1) *Generation of the functional specification.* A domain expert extracts the functional specification of each candidate reusable component from its source code and documentation. This step provides insight into the correctness of the component in relationship to the new specification. Components that are not relevant or not correct, or whose functional specification is not easy to extract, are discarded. The expert reports reasons for discarding candidates and other insights so they can be used to improve the reusability attributes model.

(2) *Generation of the test cases.* Using the functional specification, the expert generates, executes, and associates with the component a set of test cases. Components that do not satisfy the tests are discarded. Again, the reasons for discarding candidates are recorded and used to improve the reusability attributes model, and possibly the process for extracting the functional specification and assessing its correctness (step 1). This is most likely the last step at which a component will be discarded.

(3) *Classification of the component.* To distinguish it from the other components and assist in its identification and retrieval, the expert associates each reusable component with a classification according to a set of attributes identified in the domain analysis. Problems with the taxonomy are recorded for further analysis.

(4) *Development of the reuser's manual.* Information for the future reuser is provided in a manual that contains a description of the component's functions and interfaces as identified during generation of its functional specification (step 1), directions on how to install and use it, information about its procurement and support, and an appendix with structure diagrams and information for component maintenance.

(5) *Storage.* Reusable software components are stored in the repository together with their functional specifications, test cases, classification attributes, and reuser's manuals.

(6) *Feedback.* The reusability attributes model is updated by drawing on information from the qualification phase to add more measures, modify and remove measures that proved ineffective, or alter the ranges of acceptable values. This step requires analysis and possibly even further experimentation. The taxonomy is updated by adding new attributes or modifying the existing ones according to problems reported by the experts who classified the components (step 3).

This sketch illustrates the main concepts behind our approach: the use of a quantitative model for identification of components and a qualitative, partially subjective model for their qualification, with continuous improvement of both models using feedback from their application. The reusability attributes model is the key to automating the first phase.

Component identification

According to Booch, a software component "is simply a container for expressing abstractions of data structures and algorithms."[5] The attributes that make a component reusable as a building block of other, maybe radically different, systems are functional usefulness in the context of the application domain, low reuse cost, and quality.

The reusability attributes model attempts to characterize those attributes directly through measures of an attribute, or indirectly through measures of evidence of an attribute's existence. These measures must be automatable.

We define a set of acceptable values for each of the metrics. These values can be either simple ranges of values (measure α is acceptable between α_1 and α_2) or more sophisticated relationships among different metrics (measure α is acceptable between α_1 and α_2, provided that measure β is less than β_0).

Figure 3 shows a "fishbone diagram" that represents the reusability factors. With each factor in the diagram, we associate metrics directly measuring the factor or indirectly predicting the likelihood of its presence.

Costs. Reuse costs include the costs of extracting the component from the old system, packaging it into a reusable component, finding and modifying the component, and integrating it into the new system. We can measure these costs directly during the process or use metrics to predict them.

To define the *basic reusability attributes model*, the entry-level model that the component factory starts with and later improves through feedback from the qualification phase, we divide reuse costs into two groups: costs to perform the extraction and costs to use the component in a new context. To minimize the costs of finding the component and extracting it, we need code fragments that are small and simple. Measures of volume and complexity also provide a partial indication of how easy qualification will be. The costs to reuse the component can be influenced by the readability of a code fragment, a characteristic that can again be partially evaluated using volume and complexity measures, as well as measures of the nonredundancy and structuredness of the component's implementation.

Usefulness. Functional usefulness is affected by both the commonality and the variety of the functions performed by the component. The commonality of a component for reuse can be divided into three parts: its commonality within a system or a single application, its commonality across different systems in the same application domain, and its overall commonality. It is hard to associate metrics with these factors. Experience with the application domain might provide subjective insight into whether the function is primitive to the domain and occurs commonly. An indirect automatable measure of functional usefulness might be the number of times the function occurs within the analyzed system (if we assume that an often-reused component is probably highly reusable). The variety of functions performed by the component is even more difficult to measure: An indirect metric could be component complexity. However, for a component's complexity to reflect its ability to perform more functions, we would have to assume that the component was developed in a nonredundant way.

The basic reusability attributes model measures a component's functional usefulness, derived from the commonality of the functions performed by the component, by comparing the number of times the component is invoked in the system with the number of times a component known to be useful is invoked. Components known to be useful can usually be found in the standard libraries of a programming environment. The basic reusability attributes model measures the commonality of a function by the ratio between the number of its invocations and the invocations of standard components.

The basic reusability attributes model assesses functional usefulness derived from the number and the variety of functions incorporated in a component by measuring its complexity and the nonredundancy of its implementation. This last feature can be translated into volume measures comparing the component's actual volume with its expected volume, which is computed from the number of tokens (operators and operands) that the component processes. When these values are close, we say the implementation of the component is regular. High regularity suggests that the component's complexity indicates the "amount" of function it performs.

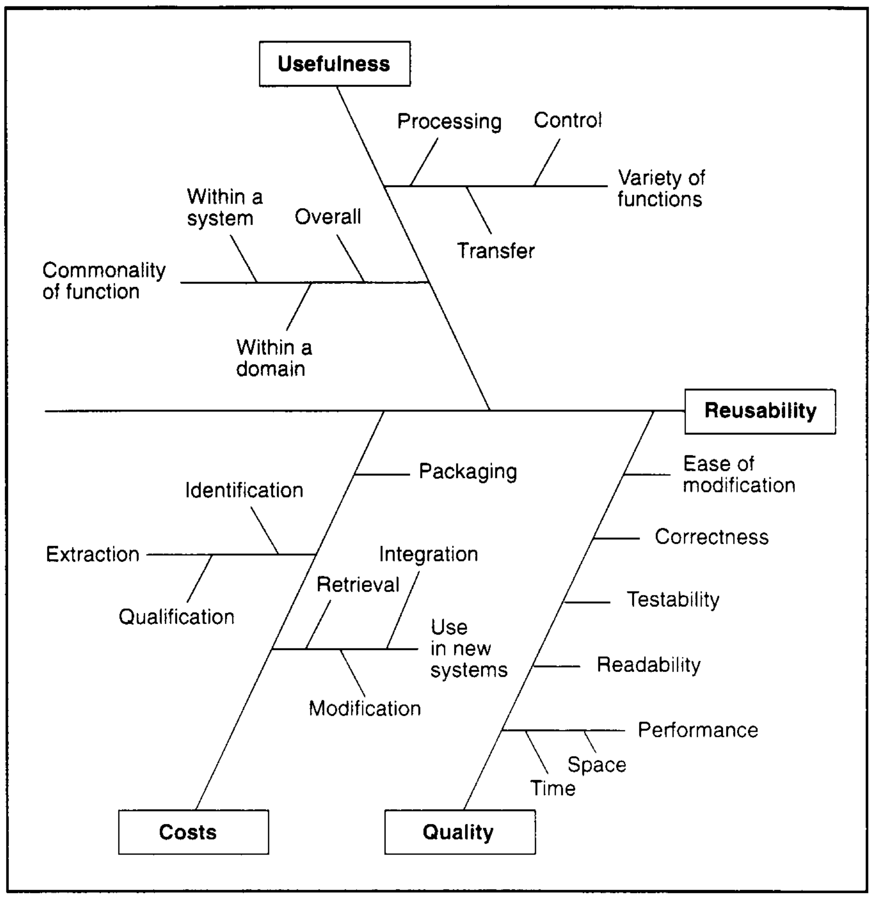


Figure 3. Factors affecting reusability.

Quality. Several qualities important for component reuse are correctness, readability, testability, ease of modification, and performance. Most are impossible to measure or predict directly. The domain expert who extracts the functional specification handles correctness and testing (steps 1 and 2 of the qualification phase). For the reusability attributes model, we are interested in qualities we can predict based upon automated measures. Therefore, we might consider such indirect metrics as small size and readability as predictors of correctness, and the number of independent paths as a measure of testability.

The basic reusability attributes model attempts to predict a component's correctness and testability using volume and complexity measures. It assumes that a large and complex component is more error prone and harder to test. Ease of modification is reflected in a component's readability.

Four metrics. Synthesizing these considerations, the basic reusability attributes model for identifying candidate reusable components characterizes a component's reusability using the four metrics shown in Figure 4.

Volume. A component's volume can be measured using the Halstead Software Science Indicators,[9] which are based on the way a program uses the programming language. First, we define the operators and the operands.

The *operators* represent the active elements of the program: arithmetic operators, decisional operators, assignment operators, functions, etc. Some operators are provided by the programming language, and some are defined by the user according to the rules of the language. The total number of these operators used in the program is denoted by η_1, and the total count of all usage of operators is denoted by N_1.

The *operands* represent the passive elements of the program: constants, variables, etc. The total number of unique operands defined and used in the program is denoted by η_2, and the total count of all usage of operands is denoted by N_2.

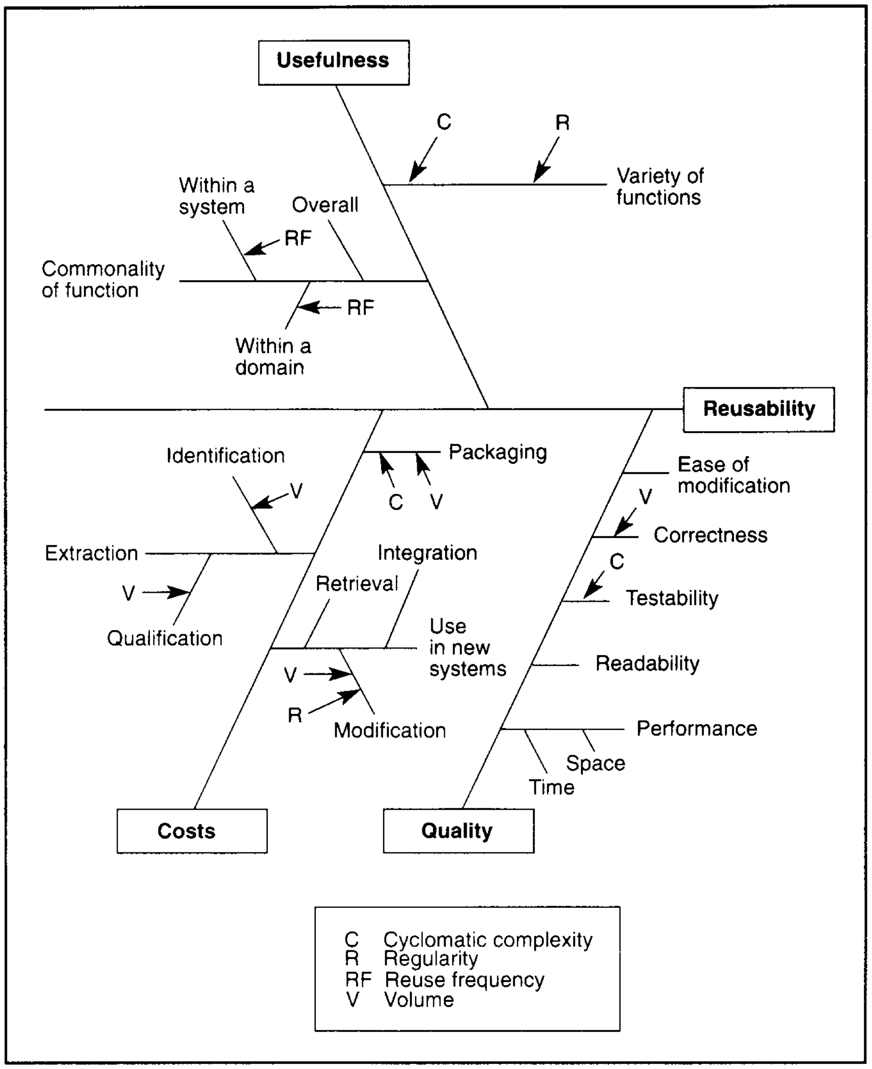


Figure 4. The basic reusability attributes model.

Using the operators and operands, we define the Halstead volume by the formula

$$V = (N_1 + N_2) \log_2 (\eta_1 + \eta_2)$$

The component volume affects both reuse cost and quality. If a component is too small, the combined costs of extraction, retrieval, and integration exceed its intrinsic value, making reuse very impractical. If it is too large, the component is more error prone and has lower quality. Therefore, in the basic reusability attributes model, we need both an upper and a lower bound for this measure.

Cyclomatic complexity. We can measure the complexity of a program's control organization with the McCabe measure,[9] defined as the cyclomatic number of the control-flow graph of the program:

$$v(G) = e - n + 2$$

where e is the number of edges in the graph G, and n is the number of nodes.

The component complexity affects reuse cost and quality, taking into account the characteristics of the component's control flow. As with volume, reuse of a component with very low complexity may not repay the cost, whereas high component complexity may indicate poor quality — low readability, poor testability, and a higher possibility of errors. On the other hand, high complexity with high regularity of implementation suggests high functional usefulness. Therefore, for this measure we need both an upper and a lower bound in the basic model.

Regularity. We can measure the economy of a component's implementation, or the use of correct programming practices, by seeing how well we can predict its length based on some regularity assumptions. Again using the Halstead Software Science Indicators, we have the actual length of the component

$$N = N_1 + N_2$$

and the estimated length

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

The closeness of the estimate is a measure of the regularity of the component's coding:

$$r = 1 - \frac{N - \hat{N}}{N} = \frac{\hat{N}}{N}$$

Component regularity measures the readability and the nonredundancy of a component's implementation. Therefore, we select components whose regularity is in the neighborhood of 1.

Reuse frequency. If we compare the number of static calls addressed to a component with the number of calls addressed to a class of components that we assume are reusable, we can estimate a given component's frequency of reuse. Let's suppose our system is composed of user-defined components $X_1, \ldots, X_N$ and of components $S_1, \ldots, S_M$ defined in the standard environment (such as printf in C or text_io.put in Ada). For a given component X, let $n(X)$ be the number of calls addressed to X in the system. We associate with each user-defined component a static measure of its reuse throughout the system: the ratio between the number of calls addressed to the component C and the average number of calls addressed to a standard component:

$$v_\sigma(C) = \frac{n(C)}{\frac{1}{M} \sum_{i=0}^{M} n(S_i)}$$

The reuse-specific frequency is an indirect measure of the functional usefulness of a component, if we assume that the application domain uses some naming convention, so components with different names are not functionally the same and vice versa. Therefore, in the basic model we have only a lower limit for this metric.

Criteria. To complete the basic model we need some criteria to select the candidate reusable components on the basis of the values of the four measures we have defined. The extremes of each measure depend on the application, the environment, the programming and design method, the programming language, and many other factors not easily quantified. We determine therefore the ranges of acceptability for the measures in the basic reusability attributes model experimentally, through a series of case studies described in the section titled "Case studies."

The basic model is elementary, but it is a reasonable starting point that captures important characteristics affecting software component reusability. Moreover, it probably contains features that will be common to every other reusability attributes model.

Care system

To support component factory activities, we have designed a computer-based system that performs static and dynamic analysis on existing code and helps a domain expert extract and qualify reusable components. We call the system Care, for computer-aided reuse engineering.

Figure 5 shows the parts of the Care system.

Component identifier. The component identifier supports source code analysis to extract the candidate reusable components according to a given reusability attributes model. The system stores candidates in the components repository for processing in the qualification phase. The identifier has two segments:

- *Model editor.* The user either defines a model, selecting metrics from a metrics library and assigning to each metric a range of acceptable values, or updates an old model from a models library, adding and deleting metrics or changing the adopted ranges of values.
- *Component extractor.* Once a reusability attributes model has been defined, the user can apply it to a family of programs to extract the candidate reusable components. The user can work interactively or extraction can be fully automated, provided that the system can automatically solve problems associated with the naming of the components.

Component qualifier. The component qualifier supports interactive qualification of the candidate reusable components according to the process model outlined earlier. For the qualifier to be effective, the candidate components must be small and simple. The qualifier has three segments:

- *Specifier.* The specifier supports the construction — through code reading and program analysis — of a formal specification to be associated with the component. The interactive tool controls as much as possible the correctness of the specification a domain expert extracts from a component. If the domain expert generates specifications, they are stored in the component repository together with the expert's measure (a subjective evaluation) of the component's practical usefulness.
- *Tester.* The tester uses the formal specification produced by the specifier to generate or to support user generation of a set of test cases for a component. If, as is likely, the component needs a "wrapping" to be executed, the tester supports the generation of this wrapping. It then executes the generated tests, reporting their outcomes and coverages. Test cases, wrapping, and coverage data are stored in the component repository with the expert's test report recommending retention or rejection of the component.
- *Classifier.* The classifier directs the user across the taxonomy of an application domain to find an appropriate classification for the component. Users with special authorization can modify the taxonomy, adding or deleting facets or altering the range of values available for each facet. The classifier and the taxonomy are directly related to the query used to retrieve the components from the repository.[6]

The current version of the Care system supports ANSI C and Ada on a Sun workstation with Unix and 8 Mbytes of memory. In the prototype, we have implemented three parts of the system:

- A component extractor for C programs based on the basic reusability attributes model described earlier. We used this part of the Care system for the case studies described in the next section. We have enriched the basic model with the data bindings metric[10] to take into account a static analysis of the flow of information between components of the same program. We have also developed a measurement tool and a data bindings analyzer for Ada programs.

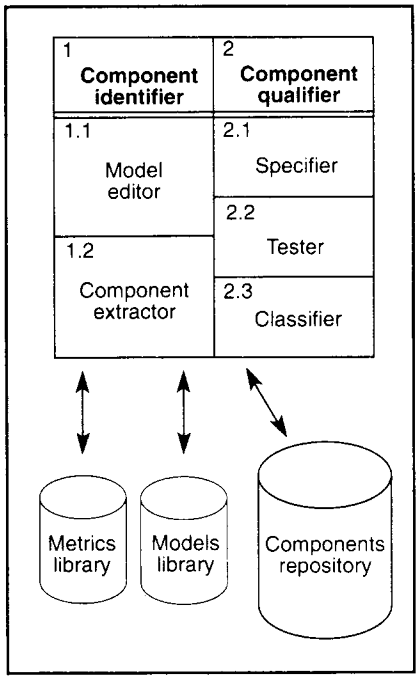


Figure 5. Care system architecture.

- A coverage analyzer for C programs (part of the tester in the component qualifier). An equivalent analyzer for Ada programs is under development.
- A prototype specifier to help the user build the Mills specification for programs written in a subset of Pascal. We plan to develop a version to process components written in C.

Case studies

In this section, we describe experiments with the current version of the Care system and the basic reusability attributes model, analyzing existing systems to identify reusable components. Some goals of the case studies were to

- evaluate the concept of extracting reusable candidates from existing programs using a model based on software metrics,
- complete the basic reusability attributes model with experimentally determined extremes for the metrics given earlier,
- study the application of the basic reusability attributes model to different environments and observe its selective power,
- analyze the interdependence of the metrics used in the basic model, and

Table 1. Characteristics of the analyzed systems.

Case	Application	Lines of Code (in thousands)	User-Defined Components
A	Data processing	4.04	83
B	File management	17.41	349
C	Communication	67.02	730
D	Data processing	17.63	156
E	Data processing	6.50	53
F	Language processing	58.55	1,235
G	File management	3.32	57
H	Communication	7.70	232
I	Language processing	4.63	87

Table 2. Average values for measures of the basic reusability attributes model.

Case	Volume	Complexity	Regularity	Reuse-Specific Frequency
A	8,967	21.1	0.76	0.05
B	7,856	23.6	0.74	0.08
C	45,707	153.7	0.66	0.10
D	11,877	32.1	0.64	0.11
E	4,054	16.8	0.76	0.18
F	82,671	198.7	0.33	0.13
G	7,277	25.5	0.65	0.24
H	12,044	40.7	0.77	0.23
I	20,131	44.7	0.79	0.41

Table 3. Measurement data for components whose reuse-specific frequency is greater than 5.0.

Case	Average Volume	Average Complexity	Average Regularity	Reuse-Specific Frequency
A	2,249	7.0	0.89	>0.50
B	2,831	4.8	0.77	>0.50
C	13,476	43.8	0.68	>0.50
D	4,444	8.5	0.80	>0.50
E	1,980	10.7	0.87	>0.50
F	156,199	384.3	0.40	>0.50
G	1,904	5.4	0.70	>0.50
H	8,884	31.1	0.75	>0.50
I	6,237	9.6	0.85	>0.50

- identify candidate reusable components to use with research and experimentation on the qualification phase.

The data we discuss here originated from the analysis of nine systems totalling 187,000 lines of ANSI C. The systems analyzed ranged from file management to communication applications, including data processing and system software. Table 1 outlines their characteristics.

Because of the characteristics of C, the natural "component" is the C function. But a function is not self-contained: It references variables, data types, and functions that are not part of its definition. To have an independent component, we had to complete the definition of the function with all the necessary external references. Therefore, in the context of the case studies, a component is the smallest translation unit containing a function. We performed each case study according to these steps:

(1) Acquire and install the system, making sure all the necessary sources are available.

(2) Build the components from the functions, adding to each function its external references and making it independently compilable.

(3) Compute the four metrics of the basic reusability attributes model for the components.

(4) Analyze the results.

Table 2 shows the average values for the measures of the basic model obtained from the case studies.

The case studies show volume, regularity, and reuse-specific frequency to have a high degree of independence. Volume and complexity show some correlation related to the "size" of the component, but it is not significant enough to make the two measures equivalent. Thus, the basic reusability attributes model is not redundant.

The data in the last column of Table 2 are below 0.5. Therefore, we can assume that a component whose specific reuse frequency is higher than 0.5 is a highly reused one. This choice is rather arbitrary, but it is useful for setting a reference point for the case studies. Accordingly, Table 3 presents the measurement data for high-reuse components whose reuse-specific frequency is more than 0.5.

Comparing Tables 2 and 3 we see, with a few exceptions, a very regular pattern. The highly reused components have volume and complexity lower than the average — about one fourth of the average. Their regularity is slightly higher than the average, generally above 0.70. The only exception is case F, a compiler with a very peculiar design, where the function calls are mostly addressed to high-level and complex modules. These results confirm, in different environments, the results obtained for Fortran programs in NASA's Software Engineering Laboratory.[11]

The regularity result is very important by itself. Because the length equation used in the regularity measure has such a good fix on the reusable components, we can use it to estimate the size of the components. Recall that the Halstead length equation ($N = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$) is a function of the two indicators η_1 and η_2. The first, the number of operators, is more or less fixed in the programming environment. The second, the number of operands, corresponds to the number of data items the system deals with. The value of η_2 can be rather precisely estimated in the detailed design phase of a project. The high regularity of the reusable components implies, therefore, that we can estimate the total effort for their development with an accuracy often higher than 80 percent. This is better than the estimate we get from components that are not as reusable.

The case studies show that, in most cases, we can obtain satisfactory results using the values in Table 4 as extremes for the ranges of acceptable values. Table 5 compares the number of user-defined functions in each system with the number of candidate reusable components extracted with the settings of Table 4.

Table 5 shows that, in general, 5 to 10 percent of the existing code should be analyzed for possible reuse. This is a cost-effective rate of reduction of the amount of code needing human analysis in the qualification phase. It is also a satisfactory figure for future reuse. In absolute terms, this 5 to 10 percent of the existing code accounts for a large part of a system's functionality.

The number of those candidates that the qualification phase will actually find to be reusable is hard to determine without a series of controlled experiments. On the basis of a cursory analysis, we think that the extracted components perform useful functions in the context of the application domain they come from. A complete and rigorous evaluation of the model is an immediate goal of our project.

Table 4. Extremes for ranges of acceptable values in the basic reusability attributes model.

Measure	Minimum	Maximum
Volume	2,000	10,000
Complexity	5.00	15.00
Regularity	0.70	1.30
Reuse frequency	0.30	

Table 5. User-defined system components compared with extracted candidates for reuse.

Case	User-Defined Components	Extracted Candidates	Percentage of User-Defined Components Extracted
A	83	4	5
B	349	17	5
C	730	36	5
D	156	16	10
E	53	4	8
F	1,235	81	7
G	57	10	18
H	232	24	10
I	87	11	13

These case studies show that reusable components have measurable properties that can be synthesized in a simple quantitative model. Now, we need to bring experimentation to the qualification activities, to verify how good the basic model is in practice, and to study how we can process the feedback from the qualification phase to improve the reusability attributes model. A possibility is a mechanism associated with the model editor for manipulating the reusability attributes model. We also need to broaden our analysis to different programming environments for broader verification of our hypotheses.

We foresee two major developments in the architecture of the Care system. The first is the design of a prototype for the components repository, supporting component retrieval both by queries to the classification system and by browsing on the basis of the specification.

The second development will be an integration with the Tame system for tailoring a measurement environment.[12] In the version of Care we outlined here, the metrics library is a static object from which users can only retrieve measures. The Tame system allows users to create a measurement environment tailored to the goals of their activities and to their model. This environment will provide a more elastic metrics library for defining measures in the reusability attributes model. ■

Acknowledgments

We are indebted to Daniele Fantasia, Bruno Macchini, and Daniela Scalabrin of Italsiel S.p.A., Rome, for developing the programs that made possible the case studies, and for many useful discussions.

This work was supported by Italsiel S.p.A. with a grant given to the Industrial Associates Program of the Department of Computer Science at the University of Maryland. Computer support was provided in part through the facilities of the Computer Science Center at the University of Maryland.

References

1. M. McIlroy, "Mass Produced Software Components," *Proc. NATO Conf. Software Eng.*, Petrocelli/Charter, New York, 1969, pp. 88-98.

2. P. Freeman, "Reusable Software Engineering Concepts and Research Directions," *ITT Proc. Workshop on Reusability in Programming*, ITT, Stamford, Conn., 1983, pp. 129-137.

3. V.R. Basili and H.D. Rombach, "Towards a Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment," Tech. Report CS-TR-2158 (UMIACS-TR-88-92), Computer Science Dept., Univ. of Maryland, College Park, Md., 1988.

4. V.R. Basili, "Software Development: A Paradigm for the Future," *Proc. Compsac 89*, IEEE Computer Soc. Press, Los Alamitos, Calif., Order No. 1964, pp. 471-485.

5. G. Booch, *Software Components with Ada*, Benjamin/Cummings, Menlo Park, Calif., 1987.

6. R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, Vol. 4, No. 1, Jan. 1987, pp. 6-16.

7. R.G. Lanergan and C.A. Grasso, "Software Engineering with Reusable Designs and Code," *IEEE Trans. Software Eng.*, Vol. SE-10, No. 5, Sept. 1984, pp. 498-501.

8. V.R. Basili and H.D. Mills, "Understanding and Documenting Programs," *IEEE Trans. Software Eng.*, Vol. SE-8, No. 3, May 1982, pp. 270-283.

9. S.D. Conte, H.E. Dunsmore, and V.Y. Shen, *Software Engineering: Metrics and Models*, Benjamin/Cummings, Menlo Park, Calif., 1986.

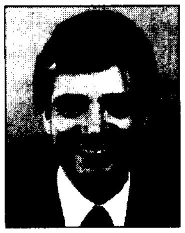
10. D. Hutchens and V.R. Basili, "System Structure Analysis: Clustering with Data Bindings," *IEEE Trans. Software Eng.*, Vol. SE-11, No. 8, Aug. 1985, pp. 749-757.

11. R.W. Selby, "Empirically Analyzing Software Reuse in a Production Environment," in *Software Reuse: Emerging Technology*, W. Tracz, ed., IEEE Computer Soc. Press, Los Alamitos, Calif., 1988, pp. 176-189.

12. V.R. Basili and H.D. Rombach, "The Tame Project: Towards Improvement-Oriented Software Environments," *IEEE Trans. Software Eng.*, Vol. SE-14, No. 6, June 1988, pp. 758-773.

Gianluigi Caldiera is on the faculty of the Institute for Advanced Computer Studies and coordinates projects on software quality and reusability in the Computer Science Department at the University of Maryland. His research interests are in software engineering, focusing on reusability, productivity, measurement, and quality management. Previously, he was an assistant professor of mathematics at the University of Rome and a lecturer in the Graduate School for Systems Engineering.

Caldiera has worked for the Finsiel Group and has been involved in the ESPRIT program as a project leader and a project reviewer. He received a Laurea degree in mathematics from the University of Rome and is a member of the IEEE Computer Society and the American Society for Quality Control.



Victor R. Basili is a professor in the Institute for Advanced Computer Studies and the Computer Science Department at the University of Maryland, where he served as chairman for six years. His research involves measuring and evaluating software development in industrial and government settings. He has consulted with many agencies and organizations, including IBM, GE, GTE, AT&T, Motorola, Boeing, and NASA.

In 1976, Basili cofounded and was a principal investigator in the Software Engineering Laboratory, a joint venture of NASA Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corp. He received the *IEEE Transactions on Software Engineering*'s Outstanding Paper Award in 1982 and the NASA Group Achievement Award in 1989. Basili is an IEEE fellow, a former member of the IEEE Computer Society Board of Governors, and a present Computer Society member.

The authors can be contacted at the Computer Science Department, University of Maryland, College Park, MD 20742, fax (301) 405-6707.