UMIACS-TR-90-30
CS-TR -2419

February 1990

Reengineering Existing Software for Reusability*

Gianluigi Caldiera† and Victor R. Basili

Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

A cost effective introduction of software reuse techniques requires the reuse of existing software, developed in many cases without aiming to reusability.

The paper discusses the problems related to the analysis and reengineering of existing software in order to reuse it. A process model is presented that analyzes the existing programs in two steps: the first one uses measurement to identify the candidate reusable components, the second one rcognizes the reusable software components and packages them into reusable units. An important characteristic of this process model is that the first step can be fully automated in order to deal with a smaller amount of code in the second phase.

The current research, ongoin at the Department of Computer Science of the University of Maryland, on this process model and on the CARE (Computer Aided Reuse Engineering) System for support of this process is presented and some experimental results are discussed.

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

†Also with Italsiel.

1. Introduction

It is commonly agreed that order of magnitude increases in productivity and quality can be accomplished only with an effective reuse of prior experience that exists in the form of knowledge, processes and products.

Reuse is a very common concept in everyday life: as P.Freeman remarks, reuse is "such a common activity, in general, that most dictionaries don't even list it", assuming that "the intelligent reader will understand that reuse means to use something again" [FRE 86]. We can transfer the idea from this everyday life understanding to the context of software engineering, and define reuse as the activity of repeatedly using existing experience, after reclaiming it, with or without modification.

In many software engineering projects reuse is as common as in everyday life: it is an informal or semi-formal kind of reuse in which information, techniques and products are shared among people working on the same or similar projects. Today software engineering research is trying to transform the informal reuse concepts into a technology of reuse, as a basis for the future software factory. The benefits of this transformation are evident to everyone, for both quality and productivity as well as for manageability of the whole production process.

Problems in achieving higher levels of reuse are the inability to package experience in a readily available way, to recognize which experience is appropriate for reuse, and to integrate reuse activities into the software development process. Reuse is assumed to take place totally within the context of the project development. This is difficult because the project focus is the delivery of the system; packaging of reusable experience can be, at best, a secondary focus of the project. Besides, project personnel are not always in the best position to recognize which pieces of experience are appropriate for other projects. Finally, existing process models are not defined to support and to take advantage of reuse, much less to create reusable experience. They tend to be rigidly deterministic where, clearly, multiple process models are necessary for reusing experience and creating packaged experience for reuse.

To address these problems, Basili [BAS 89] has proposed an organizational framework that separates the project specific activities from the reuse packaging activities, with process models for supporting each of the activities. The framework defines two separate organizations: a project organization and an experience factory.

The role of the *project organization* is to develop the product, taking advantage of all forms of packaged experience from prior and current developments. In turn, the project offers up its own experiences to be packaged for other projects.

The role of the *experience factory* is to recognize potentially reusable experience and package it in a form that makes it easy for projects to use.

One particular function of the experience factory is the development and packaging of software components. This function is performed by an organization we call the *component factory*, which supplies code components to projects upon demand, and creates and maintains a repository of components for future use. The experience manipulated by the component factory is the programming and application experience as it is embodied in programs and their documentation.

There are several activities that the component factory performs, and a set of process models associated with these activities. In the next section we will concentrate on the definition and integration of several of the process models for the project organization and the component factory with regard to the specification and ordering of components by the project, and the response by the component factory.

The rest of this paper concentrates on one specific activity of the component factory, the seeding of the repository with components salvaged from prior systems. Section 3 deals with the process model for extracting, evaluating and packaging appropriate components from existing systems. Section 4 focuses on the current reuse model for extraction and preliminary evaluation. Section 5 discusses the automation of the process model in a system called CARE (Computer Aided Reuse Engineering), the general architecture of this system, and its current implementation. Section 6 presents case studies that use the current instantiation of the reuse model.

2. A Reuse–Oriented Process Model

The project organization performs the activities that are specific to the implementation of the system to which the project is dedicated: analyzes the requirements and produces the specifications and the high–level designs of the system. The process models used by the project organization are very similar to the ones used today by software engineering projects (for instance: waterfall model, iterative enhancement model, etc.): software engineers generate specifications from requirements and design a system to satisfy those requirements. When the components of the system are identified, usually after the so called preliminary design, they are requested from the component factory.

The software objects of specific interest for our discussion are:

- *Program Units*: independent pieces of software that can be either compiled or interpreted in a given language environment: an Ada compilation unit, a C translation unit, a COBOL program, a Smalltalk class, etc.
- *Program Fragments*: a sequence of statements written in a programming language as a part of one or more larger program units: a macro, a block, a function, etc.

The elements of these two classes are what we call *software components* to emphasize their role as building blocks of larger systems. The project organization requests them from the component factory and, once they are obtained, integrates them into the programs and the system that have been previously designed.

The project organization process model, after component integration, continues as usual with product quality control (test, reliability analysis, etc.) and release.

The process model of the component factory is twofold [BAS 88]: on one side the factory satisfies the requests for components coming from the project organization, on the other it prepares itself for answering those requests. This mix of synchronous and asynchronous activities is actually typical of the process model of the experience factory in general, as pointed out by Basili [BAS 89]. Let us outline these two parts of the process model.

When the component factory receives a request from the project organization it looks up in its catalog of components to find a software component that satisfies that request with or without tailoring. There are two kinds of tailoring processes that can be applied to a software component: instantiation and modification. Instantiation has been in some way anticipated by the designer of the component, who associated with it some parameters in order to make it suit different contexts. A generic unit in the programming language Ada is an example of this kind of 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 satisfies the

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

The capability of the component factory to efficiently answer the requests coming from the project organization is a critical element for the successful application of the reuse technology. Therefore the catalog of the available components must be rich in order to reduce the chances of development from scratch, and look up must be easy. This is why there is an asynchronous part in the process model of the component factory.

Some software components are produced without specific requests coming from the project organization, developing a component production plan, generalizing components previously produced on request from the project organization, or extracting reusable components from existing systems. An example of a component production plan is provided by the Booch components [BOO 87]: the most common data structures and the main operations on them are analyzed and implemented as Ada packages. An application oriented component production plan can be performed analyzing an application domain with the aim of identifying the most common functions; then these functions can be implemented into reusable components, to be used by the developers. Generalization, on the other side, is the activity that transforms a pre-existing component into a new one, either adding more function or parametrizing it. The next sections will deal in an extensive way with extraction and reengineering of software components from existing systems.

The generated components must be well packaged and easily retrieved. This is the purpose of component qualification, an activity that adds to components functional specification, test cases and a classification based on some given component taxonomy. There are several examples of such taxonomies. G.Booch [BOO 87], for instance, classifies the components into structures, tools and subsystems, and uses, as classification attributes, the features related to concurrency, space management, garbage collection and instance visiting. Prieto–Diaz and Freeman [PRI 87] suggest a classification schema that uses facets derived from an analysis of the application domain. For instance, we can represent the functionality of a component with facets such as performed function, manipulated object, etc.; and we can represent its environment with facets such as system type, functional area, etc. A set of values for these facets provides the actual characterization of a component in this taxonomy.

Software components so produced by the component factory are stored in a repository from which they can be found when needed. The efficient organization of the repository and the ease of access to its contents are a key issue for the whole model we are outlining but we will not discuss it here [JON 88].

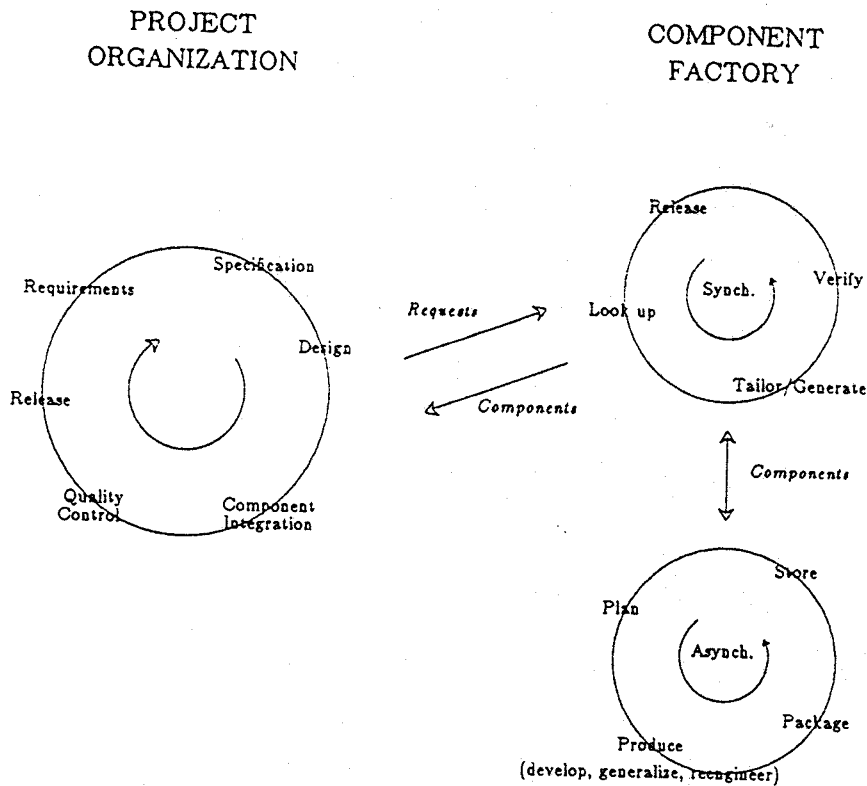

Figure 1: The Reuse Oriented Process Model

The diagram contained in Fig.1 synthesizes the three parts of the model we have described in this section.

The concepts of tailoring and generalization have been defined in the more general context of the experience factory by Basili and Rombach [BAS 88]. As they point out, both tailoring and generalization can be performed before knowing the precise reuse context (*off–line*) or after knowing it (*on–line*). The right balancing of on–line and off–line activities is very important to minimize the costs of software development in the factory and in the process as a whole. We have applied these concepts to the component factory using on–line component tailoring and off–line component generalization.

3. The Reuse Reengineering of Existing Programs

The development of reusable components is, in general, more expensive than the development of specialized code because there is a cost overhead due to the presence of the component factory. In the long term there is an economic gain from software reuse, but in the short term the cost of setting up the reuse–oriented environment may discourage many organizations from implementing the reuse paradigm. As we said in the last section, a rich and well organized catalog of reusable components is the key to a successful implementation of the paradigm. But this is not the case at the beginning, unless we are able to reuse code that was developed by the organization in the past , without having reuse in mind.

Reuse of existing code appears to be a very effective way to enrich the catalog of reusable components with a large number of items, provided that we are able to find enough components that perform useful functions. This should be the case in mature application domains where most of the functions that need to be used already exist in some form in prior systems. Those systems were designed and implemented informally reusing code. For example, rates of reuse around 60% have been found in business applications [LAN 84].

However existing code is usually not reusable as is: it must be modified and packaged for reuse. The study of this packaging can be seen as a new, growing branch of what is called software reengineering, that domain of software engineering that studies the cost–effective improvement of existing systems. Two types of reengineering are defined here:

- *reverse engineering*: the target is the system itself, that can be re–designed or simply re–documented;
- *reuse engineering*: the target is another system, that is designed reusing some knowledge, plans or products from previous ones.

Our discussion fits in the second category: we have an existing system (or several), with its source code and documentation, which we analyze in order to extract components that can be reused in implementing a new system. The reusable components are packaged and stored in a software repository available for future projects.

Reuse reengineering is therefore an asynchronous activity performed by the component factory in analyzing the existing programs. Its relevance to the activities of the component factory is very high at the beginning, when the production of a rich catalog of reusable components is a critical task. When the reuse process becomes more consolidated it becomes less relevant.

We divide reuse reengineering into two phases (Fig.2). First, we choose some candidates and package them for possible independent usage; then, we understand the service they can provide and store them in the repository with all information that has been obtained during the process. We will show that the first phase can be fully automated. The "understanding" step, instead, has to be performed

by an engineer with an effective knowledge of the application domain where the component has been developed.

The necessary human intervention in the second phase is the main reason for splitting the reuse reengineering process in two steps, when common sense would assume searching through existing programs looking for "useful" components first. The first phase reduces the amount of human analysis needed in the second phase, by limiting it to those components that really look worth considering and may be simple to understand. The second phase uses the more expensive human resources only on this reduced amount of code.

In the **component identification phase**, program units are automatically extracted, made independent, and measured according to observable properties that are related to their potential for reuse. There is a great amount of discussion about these properties: Prieto-Diaz and Freeman in their paper [PRI 87] support the idea that 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. This means we need to understand in a quantitative way the distance of the component from its potential reuse. As we will see with more detail in the next section, we propose to perform the identification phase using a quantitative model, made of a family of measures. We call it the *reusability attributes model.*

The identification phase consists of three steps:

1. *Definition (or Refinement) of the reusability attributes model.* Based upon 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. The metrics and their value ranges will be verified against the outcomes in the next steps, and continually modified until we evolve to a reusability attributes model that maximizes our chances of selecting candidate components for reuse.
2. *Extraction of components.* Modular units are extracted from existing systems and "completed" as components with all the external references needed to reuse them in an independent way, e.g., to compile them. The term "modular unit" is used here to mean a syntactic unit, like a C function, Ada subprogram or block, FORTRAN subroutine, etc.
3. *Application of the model.* The extracted, completed components are measured using the current reusability attributes model; those that pass the test, i.e., whose measurements are within the range of acceptable values for the model, become candidate reusable components that are analyzed by the domain expert in the qualification phase.

During the **component qualification phase**, candidate reusable components identified in the previous phase are analyzed by a domain expert. The goal is to understand and record the "meaning" of the component while evaluating its

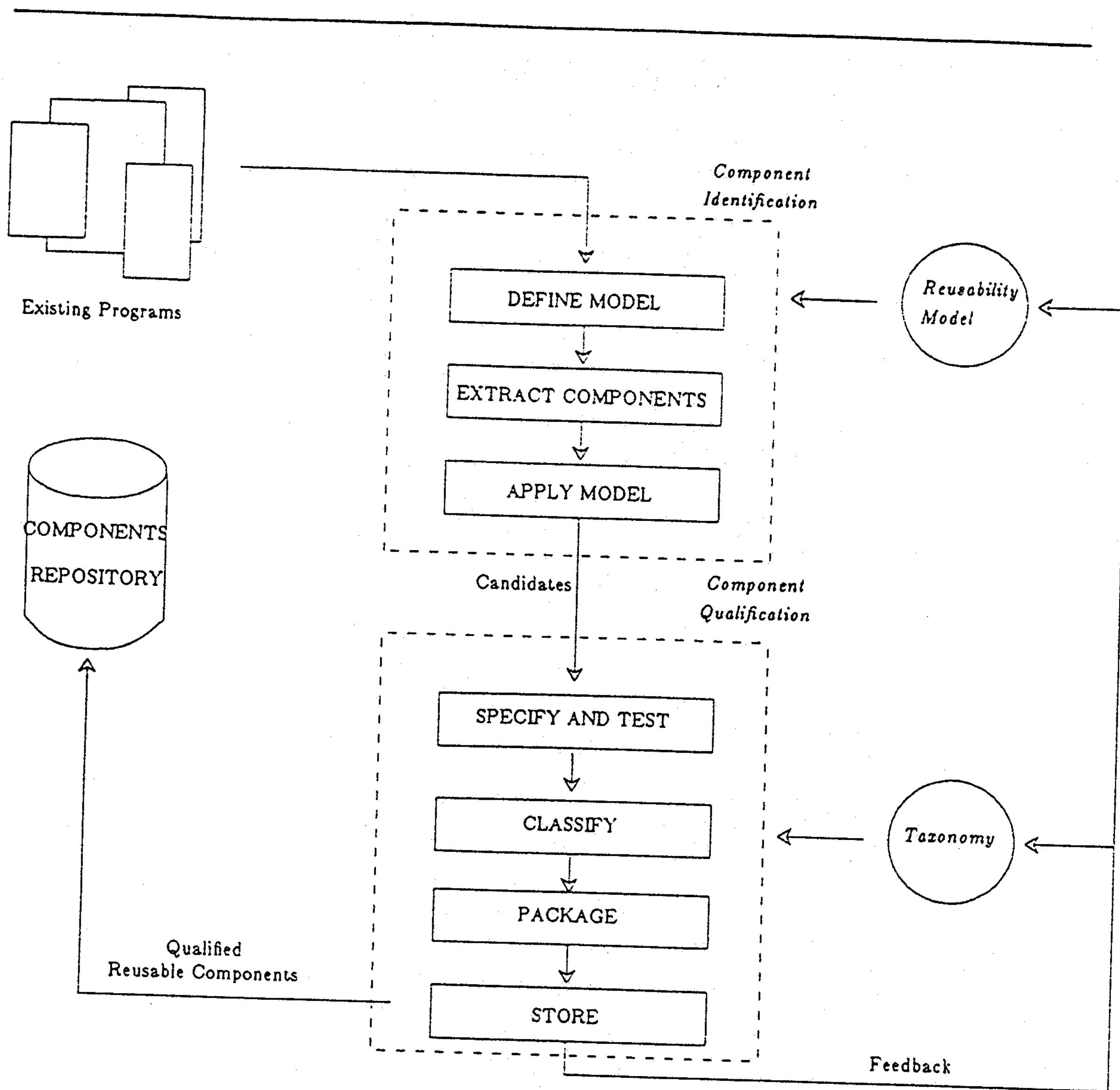

Figure 2: The Reuse Reengineering

relevance and potential for reuse in future systems, and package the component by associating a reuse–oriented specification with it. Basili and Mills have demonstrated that it is possible to understand a program and to associate with it a formal specification [BAS 82]. The technique they use can be partially automated and represents the kernel of the qualification phase. The packaging

also consists of associating with the component a significant set of test cases, a set of attributes based upon 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 changed and improved each time the qualification phase is performed by reporting and analyzing the problems that have been met.

Thus, the qualification phase consists of six steps:

4. *Generation of the functional specification.* The functional specification of each candidate reusable component is extracted from its source code and documentation by the domain expert. This step provides some insight into the correctness of the component. During this process, components which are not relevant, not correct, or whose functional specification is not easy to extract are discarded. The reasons for discarding candidates and any insights into assessing these problems during the identification phase are recorded and used to improve the reusability attributes model.

5. *Generation of the test cases.* Based upon the functional specification and other criteria for robustness and reliability, a set of test cases are generated, executed, and associated with the component. Those components that do not satisfy the tests appropriately are discarded. Again, in this step 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 correctness assessment in step 4. This is most likely the last step in which a component will be discarded.

6. *Classification of the component.* Each reusable component is associated with a classification according to a predefined set of attributes, in order to distinguish it from the other components and assist in its identification and retrieval. The problems met with the taxonomy in use are reported and will be analyzed in step 9.

7. *Development of the reuser's manual.* Information useful to the future reuser is written in the form of a manual that contains a description of the functions and interfaces of the components, 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.

8. *Store.* The reusable software components are stored in the repository together with their functional specification, test cases, classification attributes, and reuser's manual.

9. *Feedback.* The reusability attributes model is updated based upon information from the qualification phase, by adding more measures, modifying and removing those measures that did not prove effective, or altering the ranges of acceptable values. This step requires analysis and possibly even further experimentation. The taxonomy is updated adding new attributes or

modifying the existing ones according to the problems reported from step 6 of the process.

The process model we have described is just a sketch, presented to illustrate the main concepts that lay behind our approach to reuse reengineering: the use of a quantitative model in the identification phase, the use of a qualitative, and partially subjective, model in the qualification phase, the continuous improvement of both models with feedback from their application. The next sections will give a deeper perspective on the identification phase and on the reusability attributes model that plays such a key role in the process.

4. The Component Identification

A software component "is simply a container for expressing abstractions of data structures and algorithms" (G.Booch, [BOO87]). The characteristics that make it reusable as a building block of other, maybe radically different, systems are its functional usefulness in the context of the application domain, its reuse costs, and various aspects of its quality.

The reuse attribute model is an attempt at characterizing those attributes that make a component reusable, finding direct measures that evaluate that attribute, or indirect measures that provide some indication of the existence of that attribute. These measures must be automatable and are used to predict the reusability of components. Prediction is done by defining a set of acceptable values for each of the metrics. These values can be either simple ranges of values (example: measure α is acceptable between α_1 and α_2) or more sophisticated relationships between different metrics (example: measure α is acceptable between α_1 and α_2 provided that measure β is less than β_0). The acceptable values and their relationships can be represented using classification trees as proposed by R.Selby and A.Porter [SEL 89].

In the case of a component that is part of an existing program, the potential for reusability might be based upon three primary factors: the reuse cost, the functional usefulness and the quality. We can use a, so called, "fishbone diagram" to represent the influence of secondary factors on the primary ones (fig.3).

In order to build the reusability attributes model, we have to associate with every factor in the fishbone diagram of figure 3 metrics directly measuring the factor or indirectly evaluating its influence:

1. *Reuse cost* is affected by the cost of extraction from the old system, packaging into a reusable component, and the cost of finding, modifying and integrating the component into the new system. We can either measure directly these costs on the process or use metrics to predict them: there are indirect measures that contribute to the cost of packaging and modifying, such as readability, to the cost of integration, such as simplicity of the interface, and to the cost of finding the component, such as simplicity of the functional specification.

2. *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 to these factors. Experience with the application domain might provide some subjective insights into whether or not the function is

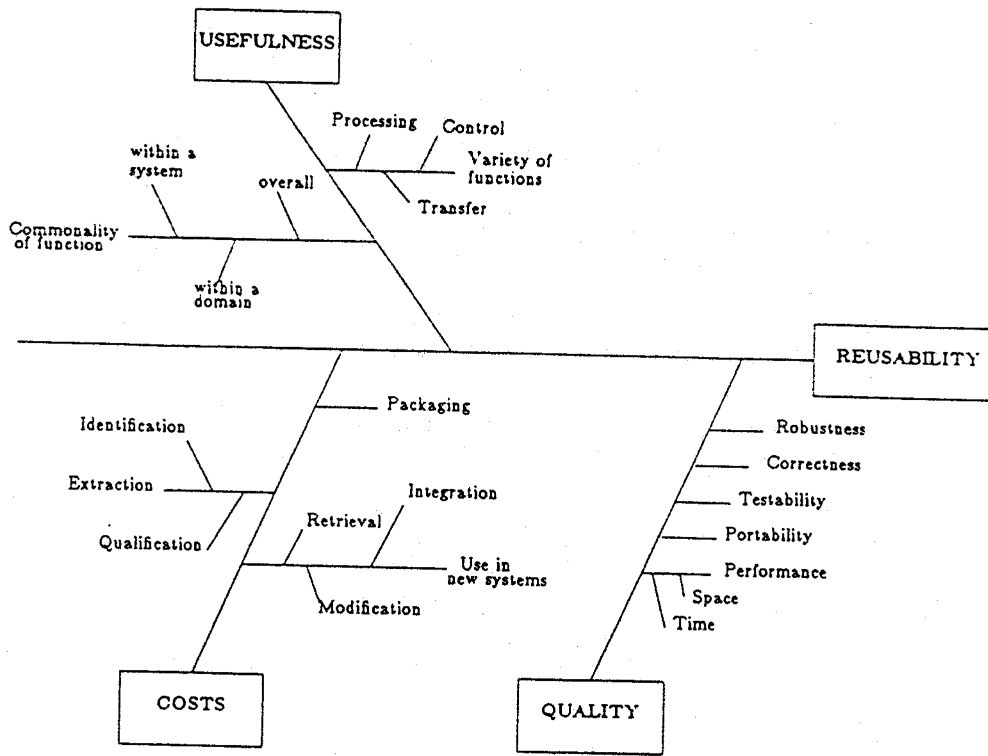

Figure 3: Factors Affecting Reusability

primitive to the domain and occurs commonly. An indirect automatable measure of functional usefulness might be the number of times the function is used within the system being analyzed, assuming that a component that is often reused is probably highly reusable. The variety of functions performed by the component is even more difficult to measure: an indirect metric can be the complexity of the component, assuming that if a component is developed in a non redundant way its complexity is higher if it performs more functions.

3. *Quality.* There are a variety qualities we care about for a reusable component. For example, correctness, readability, testability, ease of modification, performance, etc. Most of these are impossible to measure or predict directly. Correctness is handled during step 4 of the reengineering

process model and testing during step 5. For the reusability attribute model we are interested in those qualities we can predict based upon automated measures, therefore we might consider indirect metrics such as small size and readability as predictors of correctness, and number of independent paths as a measure of testability.

The component factory that extracts reusable components from existing programs must define its reusability attributes model for the identification phase and improve it through feedback from the qualification phase according to the process model outlined in the last section. In order to do this, it needs an entry-level model.

The remaining part of this section will be dedicated to the derivation and the description of such a model, called the *basic reusability attributes model.*

The reuse costs are divided in two groups: costs to perform the extraction process and costs to reuse the component. In order to minimize the costs in the first group, we need code fragments small and simple enough to make the qualification phase easier. Measures of volume and complexity provide an evaluation of this property. On the other hand, the costs to reuse the component can be represented by the readability of a code fragment, a characteristic that can again be evaluated using volume and complexity measures, as well as measures of the non-redundancy and structuredness of the implementation of the component.

The functional usefulness deriving from the commonality of the functions performed by a component can be measured by the number of times the component is invoked in the system compared with the number of times a component, that is known to be useful, is invoked. Components of this last category can be usually found in the standard libraries of a programming environment. The basic reusability model measures the commonality of a function by the ratio between the number of its invocations and the invocations of standard components.

The functional usefulness deriving from the number and the variety of functions incorporated in a component can be measured by the complexity of the component and by the non-redundancy of its implementation. This last feature can be translated into volume measures comparing the expected volume of the component, computed from the number of tokens the component processes, with its actual volume: when these values are close we say the implementation of the component is regular. If the regularity is high we can say that the complexity of the component is an indicator of the "amount" of function performed by the component.

The quality of a component can be represented by correctness, robustness and testability. These characteristics can be measured using volume and complexity measures, assuming that a large and complex component is more error-prone and harder to test. The modifiability of the component can be modeled by its readability, going back to the remarks we have made earlier in this section.

Synthesizing these considerations, the basic reusability attributes model characterizes the reusability of a component using four metrics (fig.4):

(a) *Component Volume*: The size of a component can be measured using the Halstead Software Science Indicators [CON 86]. The measure is based on the way a program uses the programming language. We define:

- the *operators*, representing the active elements of the program: arithmetic operators, decisional operators, assignment operators, functions. etc. There are operators provided by the programming language and operators defined by the user according to the rules of the programming 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

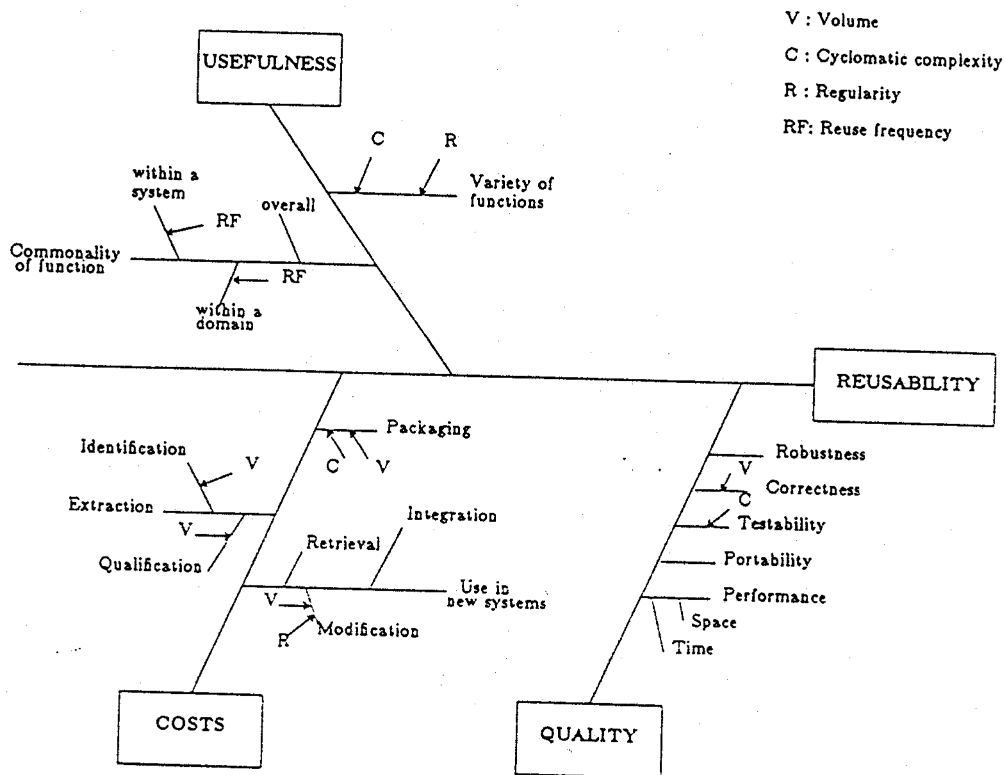

Figure 4: The Basic Reusability Model

by N_1.

- the ***operands***, representing 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.

The Halstead Volume measure is defined by the formula

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

The component volume affects both reuse cost and quality: if it 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, we need both an upper and a lower bound for this measure in the basic reusability attributes model.

(b) *Component Complexity*: We can measure the complexity of the control organization of a program by the McCabe measure [CON 86] defined as the cyclomatic number of the control flow graph G 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 control flow of the component: like volume, a small complexity affects the reuse cost, and a large one affects quality in terms of readability, testability and possibility of errors. On the other hand, high complexity, in presence of a high regularity of implementation, is a measure of the functional usefulness of the component. Therefore, for this measure, we also need both an upper and a lower bound in the basic model.

(c) *Component Regularity*: We can measure the economicity of the implementation of a component, or the use of correct programming practices verifying how well we can predict its length based on some regularity assumptions. Using, again, 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 correctness of the estimate represents a measure of the regularity of the coding in the component:

$$r = 1-(N-\hat{N})/N = \hat{N}/N.$$

The component regularity is used to evaluate the readability and the non-redundancy of the implementation of a component, therefore we select components whose regularity is in the neighborhood of 1.

(d) *Reuse specific frequency*: Comparing the number of static calls addressed to a component with the number of calls addressed to a class of components we assume reusable, we can measure the frequency of reuse of a given component. Let's suppose our system is composed of user defined components $X_1, \cdots, X_N$ and of components $S_1, \cdots, S_M$ 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:

$$\nu_\sigma(C) = n(C) / \frac{1}{M} \sum_j n(S_i);$$

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

In order to complete the basic model we need some criteria to select the candidate reusable components based on 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 on many other factors that cannot be easily quantified. We determine therefore the ranges of acceptability for the measures in the basic reusability model in an experimental way, through a series of case studies that will be described in this paper.

In conclusion, the considerations behind the basic model are elementary indeed, but the model in itself is a reasonable starting point that captures important characteristics of the phenomenon of reusability of software components and, probably, contains features that are common to every other model.

5. The CARE System

In order to support the reuse reengineering activities, a computer-based system has been designed. It performs static and dynamic analysis on existing code and helps a domain expert to extract and qualify the reusable components. The system is called CARE, for Computer-Aided Reuse Engineering. We describe in this section the system and give an account of the state of its implementation.

Modeled to support the reengineering process, the CARE System has two parts (Fig. 5):

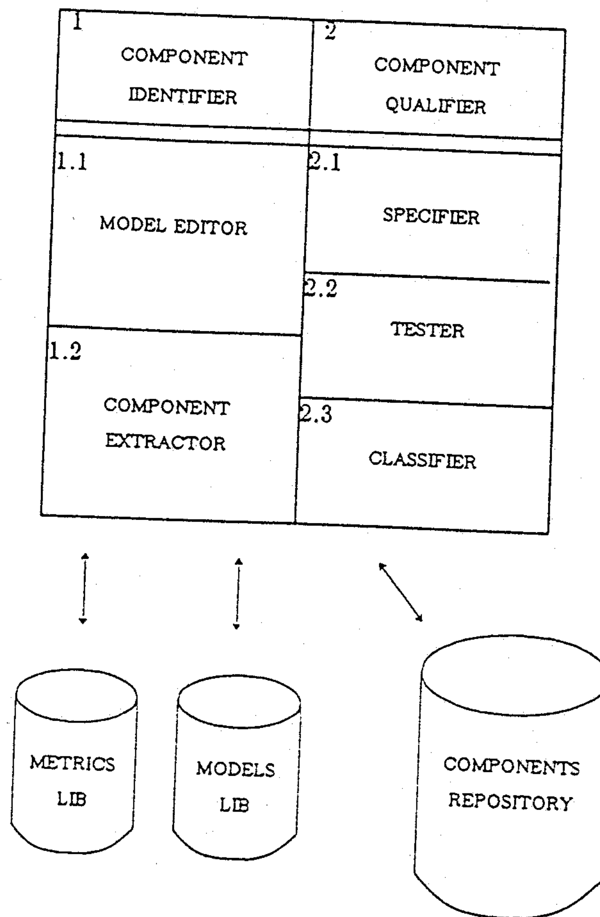

Figure 5: The Architecture of CARE

1) *Component Identifier*

This part of the system supports source code analysis to extract the candidate reusable components according to a given reusability attributes model. The candidates are stored in the components repository to be processed in the next phase. The Identifier has two segments:

1.1) *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, contained in a models library, adding and deleting metrics or changing the adopted ranges of values.

1.2) *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. This can be done either interactively by a user or in a fully automated way, provided that it is possible to automatically solve synonymy problems.

2) *Component Qualifier*

This part of the system supports interactive qualification of the candidate reusable components residing in the components library according to the process model outlined in a former section. The basic assumption behind the qualifier is that it deals with components that are small and simple enough to make qualification possible. It has three segments:

2.1) *Specifier*: It supports the construction of a formal specification to be associated with the component through code reading and program analysis. It is an interactive tool whose main purpose is to control as much as possible the correctness of the specification a domain expert extracts from a component. If the specifications are generated, they are stored in the component repository together with a measure (i.e. a subjective evaluation) of the practical usefulness of the component, given by the domain expert who is performing the qualification.

2.2) *Tester*: It uses the formal specification produced by the specifier to generate a set of test cases for a component or to support their generation. If, as it is likely to happen, the component needs a "wrapping" to be executed, it supports the generation of this wrapping (*test case generator*). Then it executes the generated tests reporting about both their outcomes and the test coverage that has been obtained (*coverage analyzer*). Test cases, wrapping and coverage data are stored in the component repository together with a test report giving an account of the grounds for keeping or rejecting the component.

2.3) *Classifier*: It directs the user across the taxonomy that is used in an application domain to find an appropriate classification for the component. Users with a 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 language that will be used to retrieve the components from the repository [PRI 87].

The CARE System is under development at the Department of Computer Science of the University of Maryland under a grant from Italsiel S.p.A., Rome, Italy. The current version supports ANSI C and Ada languages on a Sun workstation with UNIX and 8 MB memory. The parts of the CARE System implemented by the prototype are:

- the component extractor based on the basic reusability attributes model (Metrics: Halstead volume, McCabe cyclomatic complexity, regularity, reuse specific frequency) for C programs. This part of the CARE system has been used for the case studies described in the next section. The basic model is enriched, in the current version, with the data bindings metric [HUT 85] to take into account, in the static analysis, the flow of information between components of the same program. A measurement tool and a data bindings analyzer have also been developed for Ada programs.
- the coverage analyzer for C programs as a part of the tester in the component qualifier. An equivalent analyzer for Ada programs is under development.
- a prototype specifier has been developed to help the user build the Mills specification for programs written in a subset of Pascal. We are currently planning to develop a version of it to process components written in C.

Currently a prototype supporting COBOL environments is under development. This will allow us to develop more case studies in large industrial environments.

6. The LASER Project

In this section we will describe the activities of the Laboratory for Application Software Engineering Research (LASER), a research program on software reusability of the Department of Computer Science of the University of Maryland supported by Italsiel, Rome, Italy. The LASER project has developed some measurement tools, incorporated into the CARE system to implement the basic reusability attributes model and has performed many case studies analyzing existing systems to identify reusable components. The goals of the case studies have been manifold:

- evaluate the concept behind our method to extract 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 in a previous section;
- study the application of the basic reusability model to different environments and observe its selective power;
- analyze the interdependence of the metrics used in the basic model;
- identify a certain number of candidate reusable components to start research and experimentation on the qualification phase.

The data we will discuss originated from the analysis of 9 systems for a total of 186.80 KLOC of ANSI C code. The types of systems the LASER project has analyzed range from file management to communication applications, including data processing and system software.

Because of the characteristics of the C programming language, the natural candidate for the role of component is the C function, but it is not self-contained: it references variables, data types and functions that are not part of its definition. In order to have something independent, the definition of the function had to be completed with all the necessary external references. Therefore, a component, in the context of the case studies, is the smallest translation unit containing a function. Each case study has been performed 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 on the components.

(4) Analyze the results.

The average values for the measures of the basic model obtained from the case studies are presented in Table 1.

The case studies have shown a high degree of independence between volume, regularity and reuse specific frequency. Some correlation as been found between volume and complexity, both related to the "size" of the component, but not enough to consider the two measures equivalent. This means that the basic reusability model is not redundant.

The data of the last column of Table 1 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 a rather arbitrary one, but is useful to set a reference point for the case studies. With this in mind, we can now present the measurement data taken over the population of high-reuse components and compare the outcomes. The measurement data for components whose specific reuse frequency is more than 0.5 are contained in Table 2.

With respect to the goals of the case studies we outlined at the beginning of this section, the results show, with a few exceptions, a very regular pattern. The highly reused components have volume and complexity lower than the average, about 1/4 of it. 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 ones obtained in the Software Engineering Laboratory of NASA analyzing FORTRAN programs and reported by Selby [SEL 88].

The regularity result is very important by itself: the fact that the length equation has a better fit on the reusable components means that the size of such

Table 1

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

Table 2

Case	*Application*	*Avg Volume*	*Avg Complexity*	*Avg Regularity*	*Reuse Specific Freq.*
A	Data processing	2,249	7.0	0.89	>0.50
B	File Management	2,831	4.8	0.77	>0.50
C	Communication	13,476	43.8	0.68	>0.50
D	Data processing	4,444	8.5	0.80	>0.50
E	Data processing	1,980	10.7	0.87	>0.50
F	Language processing	156,199	384.3	0.40	>0.50
G	File management	1,904	5.4	0.70	>0.50
H	Communication	8,884	31.1	0.75	>0.50
I	Language processing	6,237	9.6	0.85	>0.50

components can be better estimated. The Halstead length equation ($\hat{N}=\eta_1\log_2\eta_1+\eta_2\log_2\eta_2$;) is a function of the two indicators η_1 and η_2: the first one is more or less fixed in the programming environment, the second one, 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, we can estimate the total effort for their development with an accuracy often higher than 80%, and better than the one 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 of Table 3 as extremes for the ranges of acceptable values.

Using these ranges of acceptable values in the basic model, we can see how many candidate reusable components are extracted. Table 4 compares the number of user defined functions in each system with the number of candidate reusable components obtained with the settings of Table 3.

Table 3

Measure	*Inf*	*Sup*
Volume	2,000	10,000
Complexity	5.00	15.00
Regularity	0.70	1.30
Reuse freq.	0.30	

Table 4

Case	Defined Components	Extracted Candidates	%
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 %

The data show that, in general, 5–10% of the existing code is candidate reusable, which is a good rate of reduction in terms of amount of code to be analyzed in the qualification phase, and is also a satisfactory figure for future reuse. How many of those candidates will actually be found to be reusable in the qualification phase is hard to say without a series of controlled experiments. Based upon a cursory analysis, the extracted components perform useful functions in the context of the application they come from. A real evaluation of the model is one of the immediate goals of this project.

The case studies we have described show some progress in the direction we have undertaken: reusable components have measurable properties, that can be synthesized in a simple quantitative model, like the one introduced in the last section. Now we need to bring the experimentation forward to the qualification activities, to verify how good the basic model is in practice, and to study how can we process the feedback from that phase to improve the reusability attributes model. These further case studies are currently the focus of our interest within the LASER project. We also need to broaden the spectrum of our analysis taking into consideration different programming environment to verify our hypotheses in these contexts.

7. Conclusion

The case studies we have described are just the starting point of our research on reusability. Further work is needed to assess the process model we have outlined, specifically the qualification phase. Future research will focus on the qualification phase and on the improvement of the reusability attributes model. The goal is to verify the results obtained with the basic reusability attributes model, and to improve the model taking into account these results. This means building a mechanism for the manipulation of the reusability model, associated with the model editor.

Two major developments are foreseen for the architecture of the CARE System. The first one is the design of a prototype for the components repository, supporting component retrieval both by queries on the classification and by browsing on the specification. The second will be an integration with the TAME System [BAS3 88]. In the version of CARE we have outlined in this section, the metrics library is a static object, from which an user can only retrieve measures. The TAME System allows, instead, the creation of a measurement environment tailored on the goals of the activities and on their model. This environment will be the basis for a more elastic interpretation of the metrics library, in which users will be able to define their own measures for the reusability model.

The authors are aware that much work has to be done in the field of software reusability and that this work is a small preliminary step. However, the approach outlined in this paper has the advantage of being practical and easily applied to industrial environments.

Acknowledgements

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

8. References

[BAS 82] V.R.Basili, H.D.Mills, Understanding and Documenting Programs, *IEEE Transactions on Software Engineering*, Vol.SE–8, No 3, May 1982 pp. 270–283.

[BAS 85] V.R.Basili, R.W.Selby Jr., Calculation and Use of an Environment's Characteristic Software Metric Set, *IEEE Proceedings of 8th International Conference on Software Engineering*, August 28–30, 1985, London, UK, pp. 386–390.

[BAS 88] V.R.Basili, H.D.Rombach, Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment, *Computer Science Technical Report Series*, University of Maryland, College Park, MD, December 1988, CS–TR–2158 (UMIACS–TR–88–92).

[BAS2 88] V.R.Basili, H.D.Rombach, J.Bailey, A.Delis, F.Farhat, Ada Reuse Metrics, in *Proceedings of Ada Reusability and Metrics Workshop*, Atlanta, GA, June 15–16, 1988. College Park, MD, October 1988, CS–TR–2116 (UMIACS–TR–88–72).

[BAS3 88] V.R.Basili, H.D.Rombach, The TAME Project: Towards Improvement–Oriented Software Environments, *IEEE Transactions on Software Engineering*, Vol. SE–14, No. 6, June 1988, pp.758–773.

[BAS 89] V.R.Basili, Software Development: A Paradigm for the Future (Keynote Address), *Proceedings COMPSAC '89*, Orlando, FL, September 1989, pp.471–485.

[BOO 87] G.Booch, *Software Components with Ada*, The Benjamin/Cummings Publishing Company, 1987.

[CAL 89] G.Caldiera, Software Quality and Reusability, in *Proceedings of CQS '89*, October 1989, Milano, Italy.

[CON 86] S.D.Conte, H.E.Dunsmore, V.Y.Shen, *Software Engineering – Metrics and Models*, The Benjamin/Cummings Publishing Company, 1986.

[FRE 86] P.Freeman, A Perspective on Reusability, in *Tutorial: Software Reusability*, P.Freeman (ed.), IEEE Computer Society, 1986, pp.2–8.

[GAN 86] J.D.Gannon, E.E.Katz, V.R.Basili, Metrics for Ada Packages: An Initial Study, *Communications of the ACM*, July 1986, Vol.29, 7, pp.616–623.

[HUT 85] D.Hutchens, V.R.Basili, System Structure Analysis: Clustering with Data Bindings, *IEEE Transactions on Software Engineering*, Vol. SE–11, No 8, Aug. 1985, pp.749–757.

[JON 88] G.Jones, R.Prieto–Diaz, Building and Managing Software Libraries, Proceedings COMPSAC 88, Chicago, IL, October 1988, IEEE Computer Society, pp. 228–236.

[LAN 84] R.G.Lanergan, C.A.Grasso, Software Engineering with Reusable Designs and Code, *IEEE Transactions on Software Engineering*, Vol. SE–10, No 5, Sept.1984, pp.498–501.

[PRI 87] R.Prieto–Diaz, P.Freeman, Classifying Software for Reusability, *IEEE Software*, January 1987, pp. 6–16.

[SEL 88] R.W.Selby, Empirically Analyzing Software Reuse in a Production Environment", in *Software Reuse: Emerging Technology* (W.Tracz ed.), IEEE Computer Society Press, Washington, DC, 1988, pp.176–189.

[SEL 89] R.W.Selby, A.A.Porter, Software Metric Classification Trees Help Guide the Maintenance of Large Scale Systems, *Proceedings of the Conference on Software Maintenance-1989*, Miami, Fl, October 16–19, 1989, pp.116–123.