UMIACS–TR–88–72
CS–TR–2116

October, 1988

Reusing Existing Software

Victor R. Basili
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742

Gianluigi Caldiera
ITALSIEL†

ABSTRACT

The paper examines the problems related to the analysis of existing software in order to reuse it. The source programs are analyzed in two steps: the first step is dedicated to the identification of the reusable components, the second one to their computer assisted classification. The paper deals in more detail with the identification phase on the basis of the research work in progress on a system called CARE (Computer Aided Reuse Engineering).

† This work was supported by the 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.

1. Introduction

To reuse something means to use the same thing more than once without any substantial modification to its structure.

Reuse is a common practice in many software engineering projects: it is an informal or semi-formal kind of reuse in which information, techniques and products are shared between people working on the same, or similar, projects.

According to an analysis performed on informal reuse in some environments /1/, there are three kinds of reuse:

(1) Reuse of knowledge that exists in people and documents ;

(2) Reuse of plans on how to perform certain activities, very often embodied in methodologies and standards;

(3) Reuse of tools and products.

When we speak of software reuse, therefore, we have to take into account these three types and their mutual influence. Today software engineering is developing a great deal of research in this field, transforming the informal reuse concepts into a formal technology of reuse, as a basis for the future software factory.

There are two major approaches to the technology of reuse /4/: the *composition technologies* that assume the components are atomic and unchanged in their reuse, and the *generation technologies* in which components are just patterns, either of code, or of transformation rules, having a meaning only in the context of a generator program. The main differences between the two approaches are the role, and subsequently the nature, of a component: passive in the composition approach, active in the generation approach.

We will not discuss these issues here, because both approaches are based on existing components as well as on newly designed ones. Here, we concentrate on the huge number of programs already written and available in the computer systems of every organization. A successful introduction of a reuse technology is a consequence of the benefits it brings in terms of costs and productivity: the development of reusable components is, at first, more expensive than the development of "traditional" ones, therefore the reuse of the existing code, hopefully, can balance the initial increase of costs.

We define a *reusable software object* as a piece of information that is produced in a software project and can be reused in another project. We can identify some major classes of reusable objects:

- **Architectures** : general specifications of an environment coded according to some specification/representation technique: a functional domain model, a data schema, etc.
- **Designs** : structured representations of the way a system is implemented: a program logical structure, a module interface chart, etc.
- **Tools** : sets of programs performing one or more specific functions: a compiler, a filter, a text formatter, etc.
- **Functional collections** : a set of software pieces performing related functions: a C include library, a subroutine library, etc.
- **Program units** : independent pieces of sofware that can be either compiled or interpreted in a given environment: an Ada compilation unit, a Smalltalk class, etc.
- **Program fragments** : a sequence of statements written in a programming language as a part of a larger program unit: a macro, a block, etc.

Two classes of software objects (Program units and Program fragments) are of specific interest for programming and are called *components* in order to emphasize their role as building blocks of larger systems.

Our problem, in this paper, is how we find the reusable components in the source code of an existing system, and how we make them "more reusable" by a set of suitable specifications.

We will outline a very general reuse oriented software life cycle, in order to clarify the purposes of our research. Then we will introduce the concepts of (reuse) re–engineering that are at the basis of our research. The core of the paper is the description of a process to extract the reusable components from existing source code. An implementation of those ideas is work in progress in a laboratory at the Computer Science Department of the University of Maryland under the support of ITALSIEL S.p.A., Rome, Italy.

2. A Reuse Oriented Software Life Cycle

A software engineering project reusing software components is logically divided into two major domains:

- the **business domain**, performing the activities that are specific to the implementation of the system to which the project is dedicated; the resulting software is called "business software" (the programs that are ordinarily delivered to the customers);

- the **component factory** performing the activities of implementation and maintenance of the reusable components and their tailoring to the needs of the business domain; the resulting software is called "factory software" (reusable components).

Business software and factory software have their own life cycles interacting in the implementation phases. A system is designed and its components are identified in the business domain; then the needed components are ordered from the factory. Here they are generated (i.e., retrieved or developed) and returned to the project.

For the sake of semplicity we model the business software life cycle with the traditional "waterfall" model as a sequence of steps (Fig. 1) closed by the maintenance activities. Actually many different models can be used as well.

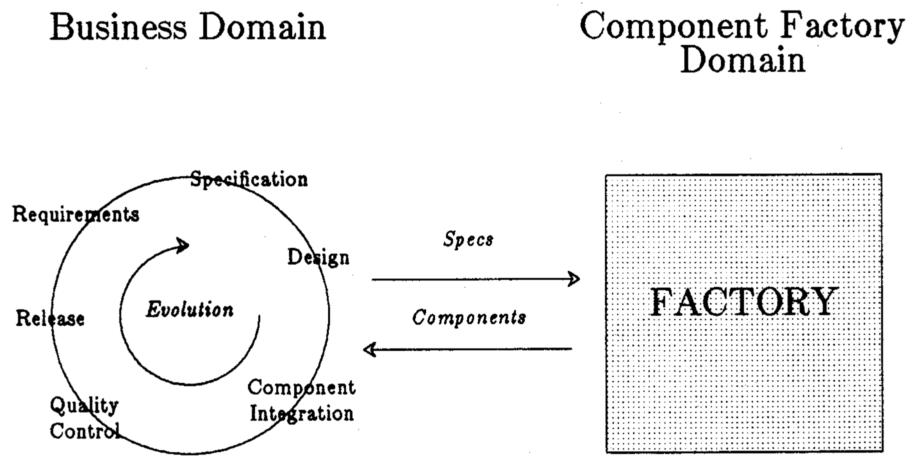

Figure 1: Business Software Life Cycle

The difference between a conventional and a reuse oriented environment comes out after the design and specification of the system. The design specs become an input to the factory that returns the implemented components.

The traditional waterfall structure doesn't change so much but the activities in the implementation phases are different. Instead of coding the programmer assembles reusable components, sometimes choosing from a set of equivalent ones returned from the factory. No change is performed by the programmer on the reusable component, no access is given to the programmer to the actual implementation of the component: the choice among different components is based on business domain criteria like performances or size.

The factory software life cycle can be modeled, for instance, on the "waterfall" scheme, but it is substantially different. A component may be implemented to satisfy a specific need of the business domain; but afterwards, is generalized and verified in order to be used again in a similar problem. Therefore we have a life cycle going from a more specific implementation to a general one, base on the concepts of transformation and generalization (Fig.2).

We can combine the two life cycles in a single model showing the interactions we have been describing. This "double waterfall" is the model informally already used in many organizations in order to reuse some software: macros, in–house developed utilities and tools, etc.

A key issue here is the way the component factory acquires and stores the components. We will not deal here with the storing of components. We will assume the factory has a "software component repository" supporting the management of the components and their retrieval according to the received specification /3/. The repository contains components coming from three sources:

- *factory development* : this is the life cycle we have already outlined, in which a component is developed from scratch in order to be returned to the business domain, then, generalized and stored in the repository;
- *external source* : the factory obtains the component from an external data repository or component vendor;

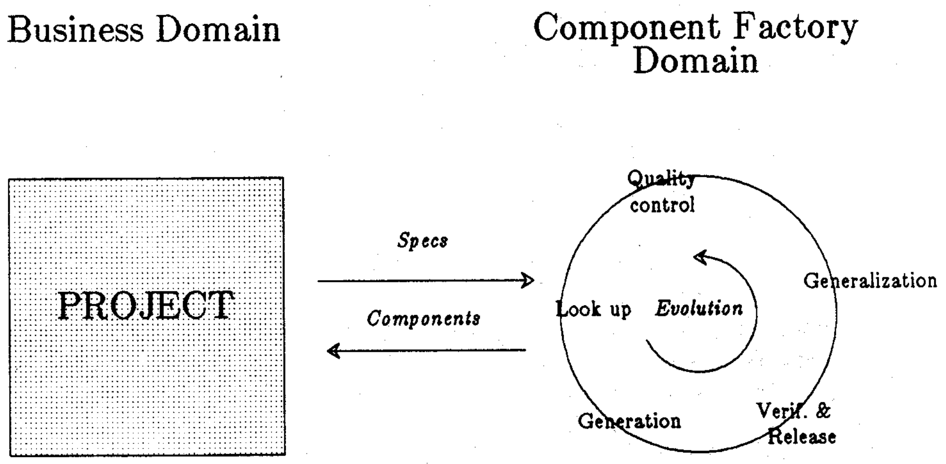

Figure 2: Factory Software Life Cycle

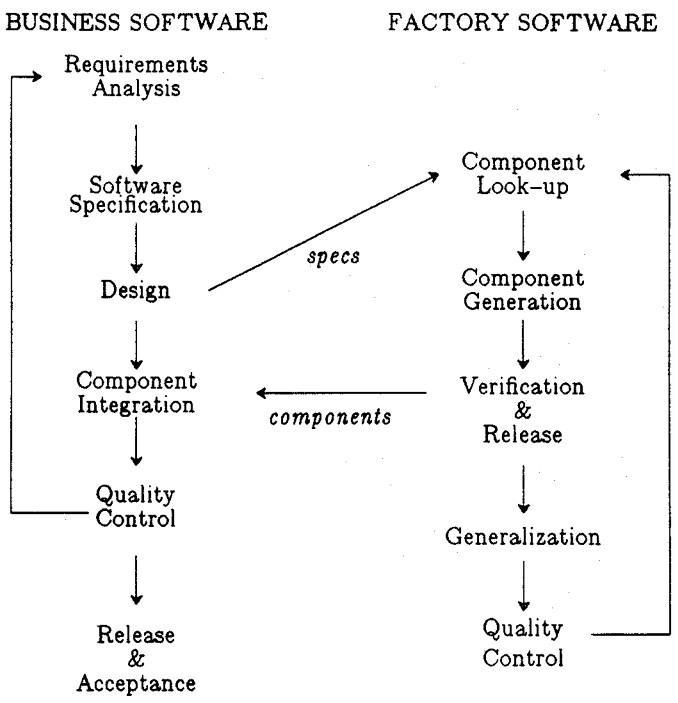

Figure 3: The two Software Life Cycles.

- *re-engineering* : the components are obtained from existing software, formerly developed with or without intent of reuse.

The factory development of reusable components is based on a design methodology aimed to reusability. Without going into the details of the problem, we can say that an object-oriented design methodology (/6/) has the needed features: rigorous information hiding, data abstraction, and modularity. In these methodologies, the elementary functions of the system are clustered within the objects, and the data flows are represented by messages exchanged by the objects. Grouping together the objects sharing common properties, we obtain the reusable components, usually called "classes", of our system.

There are many external sources and software repositories today (/7/). The Ada Software Repository is a collection of source code and of documentation accessible over the ARPA net. At the end of 1986 it contained 28 MB of source code and 11 MB of documentation.The GTE is developing a library of reusable Ada components for the US Department of Defense. Much work has been done at Intermetrics around the Ada Software Catalog (ASCAT) and the Reusable Software Library (RSL). The EVB Software Engineering Inc. sells a collection of reusable Ada components called GRACE, consisting of over 200 documented components, which deal with data structures and are based on the Brooch taxonomy /8/.

The re-engineering of a system can be one of the main sources of reusable components. The basic idea is to reverse the implementation process identifying components within an existing system with potential for reuse and qualifying them according to some reuse oriented specification technique. The description of the tasks related to re-engineering and the discussion of the feasibility of computer aided reuse engineering (CARE) are, from now on, the main interest of this paper.

3. The Re–engineering of Existing Software

The remarks we made at the end of the previous section set the context for our discussion of re–engineering. The term re–engineering denotes a large subdomain of software engineering dealing with already implemented systems, and can be used, broadly speaking, in two categories:

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

Our discussion fits in the second category: we have an existing system (or several), with its source code and documentation, which we will analyze in order to extract the components that can be reused to implement a new system. The reusable components, at the end of our analysis, will be stored in an imaginary software repository from which they will be retrieved in order tb be reused.

In defining a framework for software reuse, Basili and Rombach /12/ defined a characterization scheme that captures several aspects of the reuse process, product and environment. We can use that scheme to characterize the process which we shall outline in this paper.

The *environment* dimensions include:

(1) Application stability: How similar are past, present and future project application domains?

In our case the applications are mostly in the same domain, even though some reusable components, as system software components, can be extracted and reused in very different application domains .

(2) Process stability: How similar are past, present and future evolution processes?

We categorize such things as design methods and languages in order to judge the potential of the system components for reuse in the new system.

(3) Personnel stability: How similar are past, present and future project teams?

The main step from informal to formal reuse is the capability to reuse software without the direct involvement of the developers of the previous systems. We will introduce the concepts of reuse oriented specification in order to achieve this goal, assuming application knowledge on the part of the re–engineer.

With regard to the reuse process, there are also several dimensions based on the set of activities that are performed in reusing any object: identify and understand the components, modify and tailor them to the project needs, and integrate them into the system. The reuse *process* dimensions include:

(1) Time of activities: When do we identify, understand, modify and tailor?

This question is partially answered by the preceeding section in which we have discussed the software life cycles and the problem of acquiring and storing the reusable components. The topic of this paper, however, is only the process that identifies and understands the components.

(2) Type of modification: How much and what kind of modification needs to be made?

(3) Mechanism: How do we modify or tailor?

In the first phase of our research project, we are trying to make minimal changes to the software; therefore this is the context of this paper. Later stages will deal with varying the two last parameters /11/.

We can characterize the reuse product by taking into account the properties that are relevant for its reuse. The reuse *product* dimensions include, therefore:

(1) Type: What is a characterization of the object?

We focus our attention on software components, as they have been characterized in a preceeding section.

(2) Usability: How independent and understandable is it?

(3) Quality: How good is it?

Our process is mainly dedicated to answer the last two questions through a combination of syntax analysis, measurements and specification techniques. The properties of the software components are taken into account separately and then related to each other according to a so called reusability attributes model.

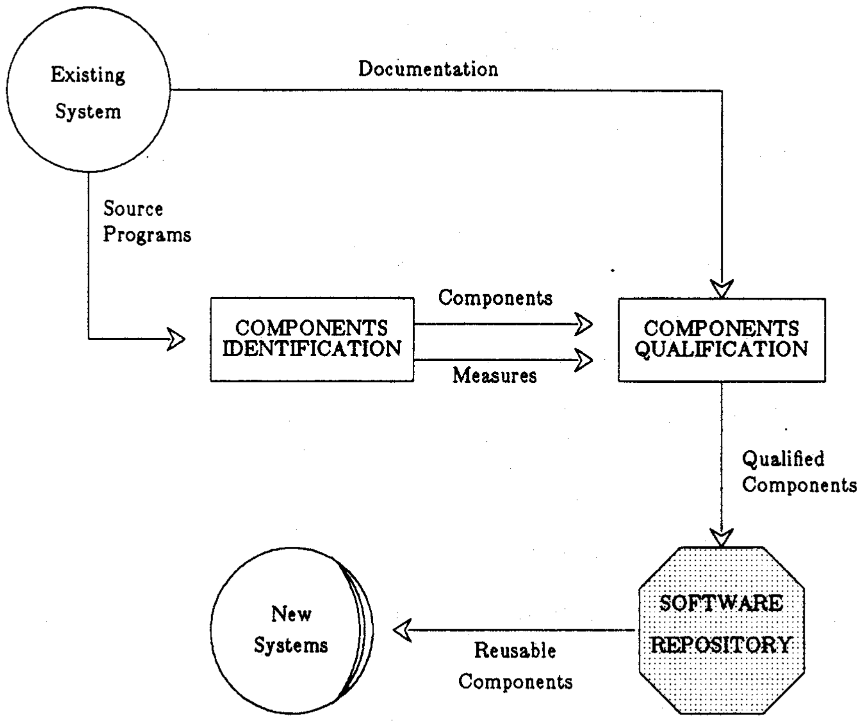

Figure 4: The Phases of Re–engineering.

We define two phases, by distinguishing what can be fully automated (the first phase) from what needs some interaction with an application domain expert (the second phase).

The first phase is the **components identification**: it analyzes the source code of the existing system, trying to acquire from it as much information as possible. It can be automated to generate reuse candidates, because its input is written in a formally specified programming language. It uses several measures to quantify the observable properties of a software component that are in some way related to its reusability; and, it matches the results of the measurements with what we think is a "reasonable" profile for a reusable component. There is a great amount of discussion today about the characteristics that make a component reusable /2/. Prieto–Diaz and Freeman in their paper /5/ 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 that a reusable component has a small size and a simple structure, is accompained by an excellent documentation, and is written in a suitable programming language. Our measures will try to quantify these intuitive ideas. The product of the components identification phase, as outlined in Fig. 4, is a set of components with the related data, describing their reusability and their actual reuse in the existing system.

The second phase is the **components qualification**: it analyzes the reusable components identified in the previous phase in order to understand their "meaning" and to associate them with a reuse-oriented specification. This process of understanding the meaning combines what can be derived from the component with the domain knowledge owned by the software engineers and contained in the specifications of the system. Basili and Mills have demonstrated in /14/ that it is possible to understand a program and to associate with it a formal documentation. The technique they use can be partially automated and represents the kernel of the qualification phase. The outcome of this phase is a set of specifications that is associated to each reusable component and, as we will see, one that allows its classification according to a general framework for the classification of software components (*taxonomy*). There are several examples of taxonomy: G.Booch /8/, 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 /5/ represent the functionality of a component by three attributes <function, object, medium> and its environment by three more attributes <system type, functional area, setting>.

After this overview of the process, let's go into a more detailed discussion of the two phases of reuse re-engineering.

4. Components Identification

According to our previous definition, a software component, both a program unit and a program fragment, is an aggregate of data structures and algorithms. In order to be reusable, it must be enough independent and modular to be thought as a building block of other software components. This means we expect to have

- *compact components* defining and manipulating only one kind of object;
- *independent components* usable without requiring the presence of other ones.

For instance, a component that manipulates bounded stacks should not manipulate binary trees; and, if this is the case, we should be able to split the component into two compact ones, based on the two different data types they manipulate. Besides, we want to be able to manipulate bounded stacks using only that component.

The reason to require compact components is that non-compact ones are difficult to qualify and impractical to use. However, compactness should be achieved without loosing independence: when we use a component we should be able to use it as a self contained object, without the need of other components. The component itself must encapsulate all the computational and manipulative features that are needed to use it. The two properties appear in contradiction with each other but they are not: compactness requires components to have a conceptually simple specification and independence requires a self contained implementation.

If we can analyze an existing system, we can measure the frequency of reuse of some components: this is important information about their reusability, because *a component that is often reused is probably highly reusable.*

The identification process therefore is made of two steps:

(1) The source code of the programs of the existing system is analyzed in order to extract the subprograms and to understand their relationships; this step produces a list of *candidate components*.

(2) The candidates are analyzed using a reusability attributes model, and the actual *candidates reusable components* are selected; this step produces a list of reusable components that will be submitted to the specification process.

The *reusability attributes model* is the formalization of the properties that are pertinent to the reuse of a software component. It is composed by measures and by exogenous constants, where the first ones are a quantification of the intuitive properties, and the second ones are the ranges of values, inside which we consider a component to be reusable. Properties like compactness and independence are in the domain of general software engineering, so to speak, and can be approached using general purpose measures like cyclomatic complexity, software science measures, data bindings, package visibility, component access, and any other measures that provide insight into the quality of the component. Other properties, like the frequency of reuse, are more specific of the re–engineering domain, and are focused on reusability.

The reusability attributes model, therefore, is composed by two sets of measures: measures related to general properties of the components $\mu=(\mu_1,\mu_2,...,\mu_k)$ and measures related to the reuse specific properties $\nu=(\nu_1,\nu_2,...,\nu_k)$. The model associates these two sets of measures with two sets of exogenous constants α_μ and α_ν that represent the ranges of values for those measures. The term "exogenous" means that the values of the constants are set from the outside in order to have a better fit between reusability attributes model and reality.

The presence of the exogenous constants allows our model to be tailored to every environment, and to be improved with its use. We can start to use the model with a generic setting for the variables; then, using the model, we can modify that setting in order to reach a better fit with the environment of our project.

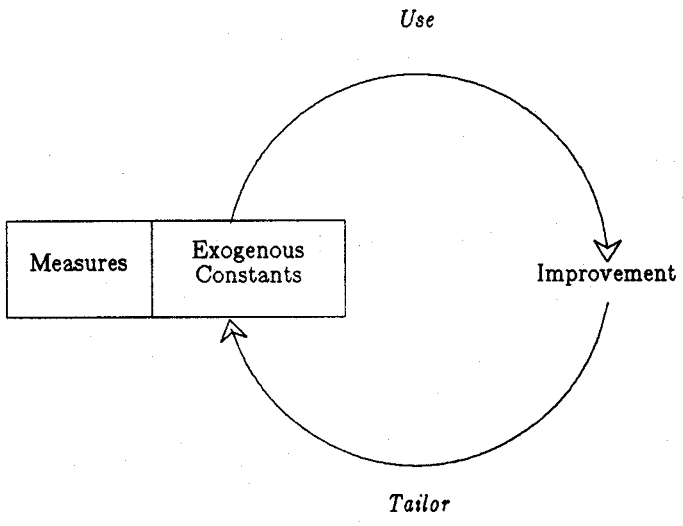

Figure 5: The Reusability Attributes Model and its Improvement

We select the components according to computations based on the model, but the model itself is tuned to the environment. Abstract prescriptions, in order to identify the components from the subprograms in a given environment, cannot be given: the model is extendable, can be tailored to the environment, and improved with use (Fig. 5). This approach responds to the need of supporting the reuse analysis with the existing experience and tailoring it to the software engineering environment /12/.

The implementation of the model we are currently building is an interactive system in which a set of tools analyzes the source programs and computes the metrics. If the user is not satisfied with the parameters of the model he is currently using, he can change them interactively, regenerate the model and run it again. Browsing and navigating capability are offered to the user for an easier operation.

4.1. Components Extraction

The first step of the component identification is based on the syntax of the programming language used in the analyzed system. We will use C and Ada in our examples, but the discussion is not restricted to those environments, as long as the programming language is structured, we could easily use Pascal or Modula-2.

The way the components appear in a program is determined by the programming language. If we have a C program like this:

```
main () {
      printf ("Hello World!\n");
      }
```

we recognize that the program is using the component "printf", because we know that the C language invokes the components as functions operating on arguments. The component actually being used depends on the visibility rules of the environment: the function "printf" must be contained in a library that is visible from the program which uses it (the include library "stdio.h").

An analogous example can be made in Ada:

```
with text_io; use text_io;
procedure hello is
begin
      put("Hello World!");
      new_line;
end hello;
```

Here the components are grouped together into packages ("text_io", in the example), and the context is explicitly declared in the source code.

Both C and Ada deal with components in the same way: declaring them and invoking them. *Components are identified by their name,* sometimes extended by the name of the package containing it. There are of course problems related with the enforcement of naming conventions in the environment we are analyzing. If enforcement is weak we may have different groups of people working on the same project with inconsistent nomenclatures:

- identical components having different names ;

- different components having the same name.

An analysis based on the syntax of the programming language cannot solve the intricate problems that may rise in this context. The interaction with the user, outlined in the preceeding section, is the key for the solution to the major problems rising in this area.

Abstracting from the programming language /9/, we can outline the structure of a general program by identifying declarative sections and procedural sections. The declarative sections define the objects that are going to be used in the procedural sections: types, variables, subprograms. The last ones are programs themselves, and have the same structure. The procedural sections contain the statements that invoke the subprograms, and, in some cases, more declarative sections.

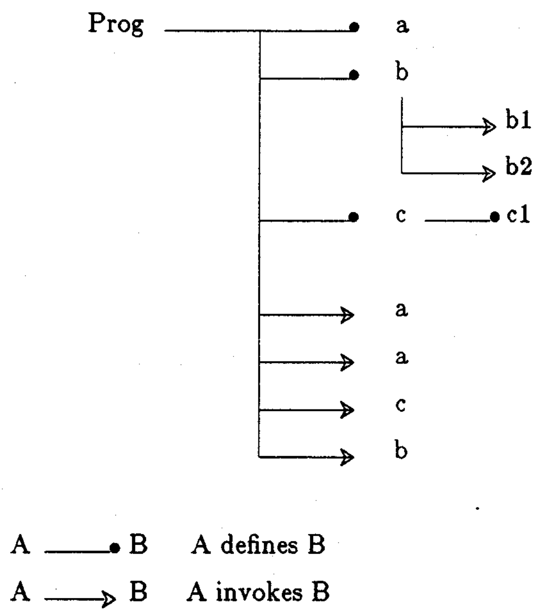

Figure 6: The Subprograms Map Tree

For every source code file written in a given programming language, we can generate a map of the subprograms in that file. We have therefore a Subprogram Map Tree for each program unit: the nodes represent the subprograms and are labeled with their extended name (container name + program name); the branches represent either the definition or the invocation of a subprogram unit. The subprograms listed in the tree are our candidates as reusable components: they form a broad list from which we are going to select the elements that show some reusability.

4.2. Components Selection

The Subprogram Map Tree is a structured set of candidates obtained by a syntactic analysis of the source file we have analyzed. In order to select the reusable components from this set of candidates, we use the reusability attributes model introduced in a previous section. The implementation of this model is a filter applied to the Subprogram Map Tree cutting some leaves when the measured quantities are far from the values given by the exogenous constants. The candidate reusable components are the result of this selection (Fig. 7).

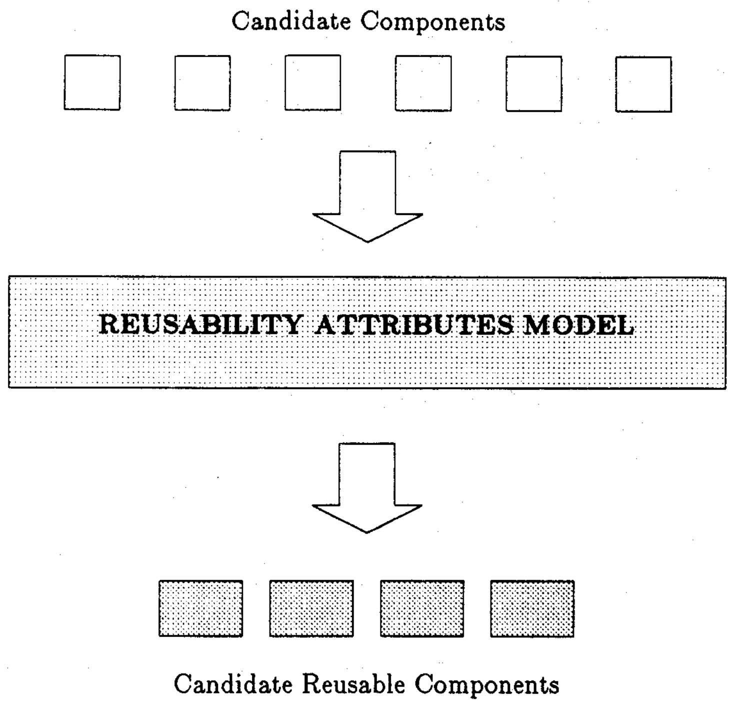

Figure 7: The Components Selection

Let's discuss now the measures contained in the reusability attributes model according to the distinction between general measures μ and reuse specific measures ν. Both sets are an initial collection of measures being implemented for the first prototype. It is assumed that metrics wiil be added, deleted or modified, based upon experimentation. The reusability attributes model will evolve and vary with the sub-application domain, language, etc., and, hopefully, will reach a characteristic set of metrics (/22/).

The first two measures we introduce are the data types measure μ_{dt} and the data bindings measure μ_{db}

If a subprogram defines a data type and one of its subprograms deals with it, it is necessary to have "visibility" of that definition in order to use the subprograms as independent components. Let's see an Ada example:

```
procedure A( ... ) is
  type T1 is new ... ;
  ...
  procedure B( x: in T1; ... ) is
  begin
    ...
  end B;
begin
  ...
  B(x0; ...) ...;
  ...
end A;
```

If the number of nesting levels between the subprogram A containing the data type definition, T1 in the example, and the subprogram B using the data type is small, we assume the type is tailored to be used by that subprogram and group together the subprograms A and B in a single component. The concept of "small" is a bit fuzzy and depends on the granularity of the design: we define therefore an exogenous variable to define its extremes.

The *data types measure* μ_{dt} is applied to a data type T and a subprogram S referencing T, and is the number of nesting levels between subprogram P defining T and the subprogram S using T. The exogenous constants $\alpha_{\mu_{dt}}^{(1)}$ and $\alpha_{\mu_{dt}}^{(2)}$ represent the extremes of μ_{dt}: if $\mu_{dt}(T_P,S)$ is inside the range defined by $\alpha_{\mu_{dt}}^{(1)}$ and $\alpha_{\mu_{dt}}^{(2)}$ we define P as a single component including S.

A similar analysis of the clustering of subprograms into components can be performed using the data bindings measure /10/: let x be a variable and A and B subprograms, if A assigns x and B references x the triple (A,x,B) is a data binding among A and B. Data bindings represent a communication path between subprograms that can be evaluated quantitatively. The application of data binding analysis to reuse of Ada code has been discussed in /11/, and we will not repeat that discussion here.

We define therefore a *data bindings measure* μ_{db} and associate with it, as we did for the data types, an exogenous constant $\alpha_{\mu_{db}}$.

The reusability attributes model contains other general measures that quantify properties of a software component relevant to its potential reuse /5/. They are, in our model,

1) *Program Size* measured by the number of lines of code μ_{LOC} and by the Halstead length /13/ of the component μ_N.

2) *Program Structure* measured by the number of subprograms and links and by the McCabe cyclomatic complexity μ_{CYC} of the flowgraph representing the program /13/.

A small and simple software component is easier to deal with, if we want to reuse it. As Prieto–Diaz and Freeman /5/ observe, these factors are modified by the experience of the reuser, who has its own criteria to decide what is small and simple. In order to take into account these criteria, we can combine the measures L and $\nu(G)$ with a subjective evaluation of the size and complexity of the components, obtained from the staff of the project.

3) *Programming Language* measured by the Halstead language level indicator μ_λ /13/.

The μ_λ indicator doesn't say that a language is "better than" another one: it just measures the readability of a component written in that language.

4) *Program Documentation* measured by a combination of the language level for the specification language (if any) and by a subjective rating.

The documentation we are referring to is the one that is ordinarily associated with the source code of a program. We will see, in next section, that there are several kinds of documentation we can attach to a component, but this one is the only one that is available at the beginning of a re–engineering activity. If the purpose and the interface of the component are well defined, the component is more reusable.

5) *Program Reliability* measured by a combination of the expected number of errors and by the number of reported errors in the component, or by some more sophisticated reliability measure /21/.

The measures we have introduced are just some relevant ones and any number of measures can be added to the *measurement vector* μ. The exogenous constants α_{μ_i} represent again the limit values of the measured properties, in the environment where we are using the model. These limits will be modified based upon experience.

Let's now describe a set of more specific measures of the reusability. They are basically frequencies normalized over the total set of reuse events we can see in a static environment. It would be interesting to have data about the dynamic reuse of the components, i.e. how many times components are actually reused by the running system. This is one of the goals of our future research.

As we said in a foregoing section, a component that is highly reused is probably highly reusable: a direct consequence of this simple remark is that we have to evaluate the actual reuse of a component in order to establish how significant the measures $(\mu_1,\mu_2,\dots,\mu_k)$ are. Besides, a high reuse rate is itself a reusability measure, because the component is well known throughout the project and the engineers know it is available and rely on it. The *reuse (measurement) vector* ν is therefore attached to each component like the measurement vector μ.

We initially define two reuse measures for each component:

1) *Reuse frequency*: the ratio between the number of calls addressed to that component and the total number of component calls in a given collection of programs:

$$\nu(C_0)= n(C_0)/ \textstyle\sum_i n(C_i)$$

where C_0 is the component whose reuse we are measuring, and C_i are the components called in the given collection of programs. This is therefore the frequency of reuse of the component C_0 calculated over the global reuse of all components $\{C_i\}$ in the system.

2) *Reuse specific frequency*: the ratio between the number of calls addressed to that component and the number of calls addressed to a choosen component C_p in the standard environment of the program (for instance: stdio.fprintf in C, text_io.put in Ada):

$$\nu_\sigma(C_0)= n(C_0)/ n(C_p)$$

where the choice of C_p is made according to the characteristics of the collection of programs, selecting an often used component. This is the compared frequency of reuse of the component in relationship with the reuse of a "special" component.

Both reuse metrics are calculated on a given collection of programs. There are many ways to choose this collection: a particularly representative subset of the programs in the system, the whole system, a random sample of the programs in the system, etc.

The exogenous constants α_{ν_i} are here the minimal reuse frequencies, under which a component is not reused enough to be considered reusable by the model. The necessity of adapting these values to the context and choosing carefully the collection of programs on which to compute the measures, is evident.

5. Components Qualification

We will not discuss here the components qualification in the same detail as components identification: it will be the topic of a future paper. Therefore the goal of this section is to complete the picture of re–engineering, adding some hints about the second phase.

Qualifying a component means specifying it and find its place in a given component taxonomy. There are two kinds of relevant specifications in our context: the *functional specification* saying what the component does and the *interface specification* saying how the component can be connected to other ones to perform more complex functions.

From these two kinds of specifications we can derive the *classification* of the component according to a given taxonomy, i.e. a short, structured summary of the meaning and of the role of the component, mainly suitable for retrieval purposes.

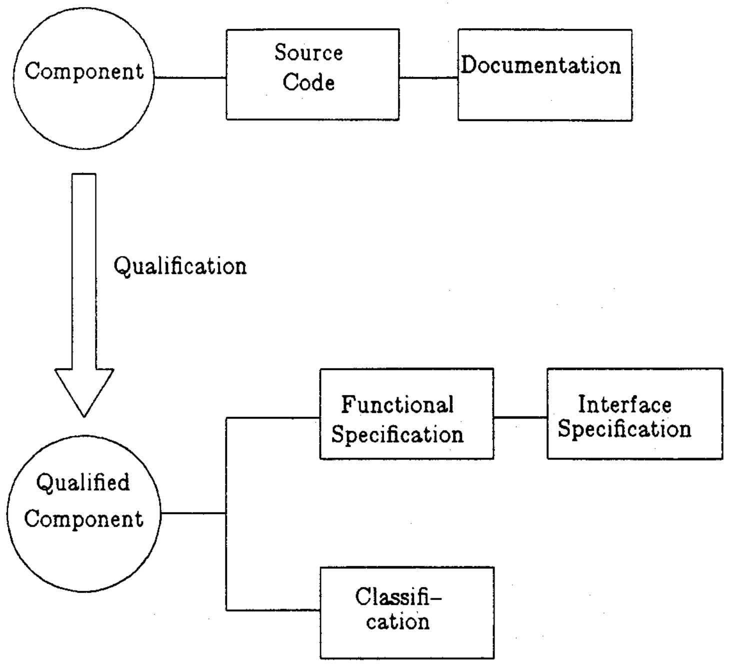

Figure 8: The Component Qualification

The two specifications and the classification, attached to every reusable component, are its *reuse-oriented specification* (ROS) and will be stored with it in the software repository.

There are many languages that can be used to write the ROS of a component and the references in this paper give an account of some related literature (/15,16,17,18/).

The topic we want to deal with is the *classification of components*. A simple and very useful taxonomy is discussed by Prieto–Diaz and Freeman /5/ and produces a faceted classification: two groups of attributes are attached to each component, and represent its functionality and environment. The version we give here is a slightly modified one. We classify the components according to a semantic data model representation of the application system in which they are used.

Every data processing system can be represented by a model written according to a choosen modeling language. A very common one, which represents the conceptual objects by entities and relationships and their properties by attributes appended to them /19/, is called Entity–Relationship model. There is definitely a finite number of operations that can be performed on a conceptual object or on its attributes: therefore, when we want to classify a component assigning to it a functionality, we just have to transform the "intended function" contained in the functional specification into one of those operations. The first two attributes of the classification are: <Object, Function>. Very often it is interesting to know the underlying structure that implements a conceptual object: an external data storage, a table, a message, a node of a distributed system. This provides a third attribute, called <Medium>, for the functional classification of a component. The following table contains some examples of the values for the attributes.

Functional Attributes		
Object	Function	Medium
Entity 1	Add	Storage
Ent1.Attribute1	Modify	Workstation
Ent1.Attribute2	Join	Sensor
...	Decode	Stack
Entity 2	Compare	Queue
Entity 3	Append	Tree
...	Close	Mouse
Relationship 1	...	...
Relationship 2		
...		

These attributes can be partially derived from the functional specification of the component and partially provided by a software engineer knowing the application domain. The existence of the conceptual schema is essential for the classification of the components and is one of the goals of the domain analysis.

The environment of a component is divided into intrinsic environment, stating the program's interfaces with the external world (how many and what kind of "ports"), extrinsic environment, stating the functional area (the name of the project in the business domain), and operating environment. The following table contains some examples of these attributes.

Environment Attributes			
Port		Functional	Operating
Number	Type	Area	Environment
P1	Ascii File	...	MS-DOS
P2	Interrupt		UNIX
P3	Binary File		MVS DB2
...	Binary Packet		...
	...		

The data for the environment attributes can be, once more, obtained either from the interface specification or from the interaction with a software engineer.

The complete collection of classification attributes associated with every component is the search key for every component search in the data repository, with a procedure similar to the one described by Prieto-Diaz and Freeman /5/. It allows us to go back to the identification phase when we recognize similarities among components having different names. The interaction with an engineer is here crucial: the system will list the components having the same classification pointing out that they might be the same thing, then it is up to the engineer to decide if they are really the same despite the different appearance.

6. Conclusions

We have shown that the outcome of the re-engineering process, as we defined it, is a collection of reusable components ready to be stored in the software repository together with all the information we have extracted from the system during the two phases described. The components factory will access the repository to retrieve and update that information, and will mantain the integrity of the system.

Two are the major benefits we see in using the outlined techniques: the immediate availability of a large number of reusable components to be used in the implementation of new systems, and a better understanding of the functions and the structure of those components. This last point is very important because the information we obtain from the analysis of the reusable components can be "reused" in designing new ones.

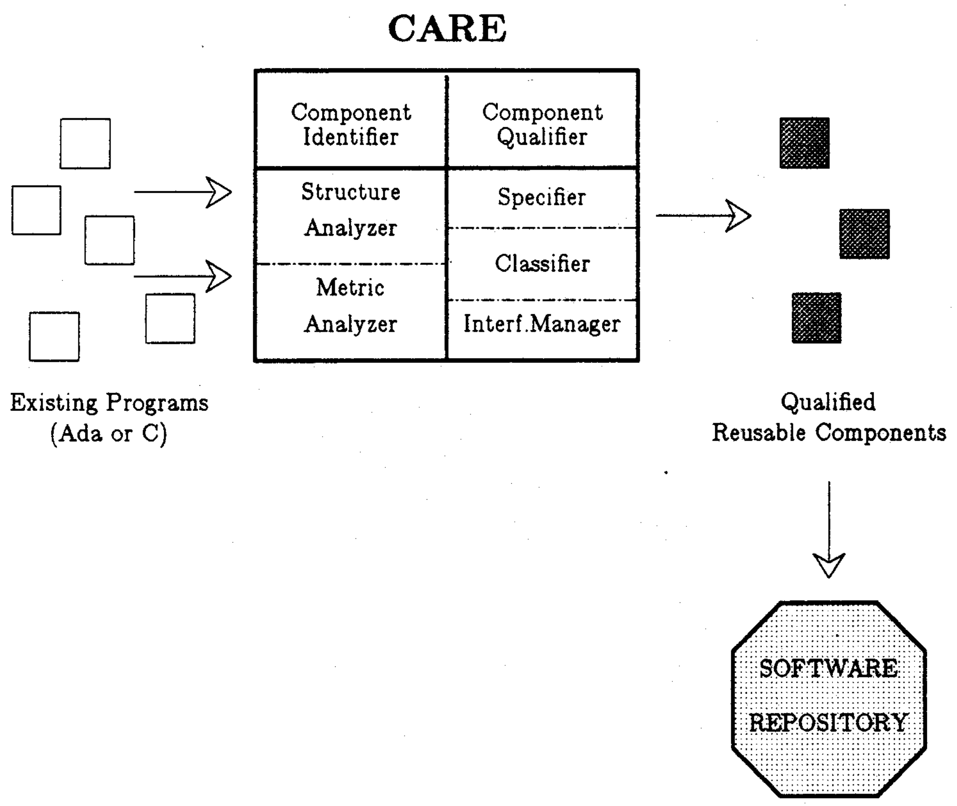

Figure 9: The CARE System

The Computer Science Department of the University of Maryland and the ITALSIEL S.p.A. are currently working to a first prototype of the **CARE system**: it will be able to perform the re-enginnering activities on both C and Ada programs (Fig. 9). The system will be able to identify and measure the reusable components in an existing application domain, and to qualify them with the help of a domain expert. The first version of the prototype is expected by the first half of 1989 and will by tested on a case study from the MIS area. It will

- extract and select the components from a set of programs;
- measure their reusability and frequency of reuse;
- restructure the selection according to those measurements;
- analyze the statistical correlation between the μ and ν vectors.

A second prototype of the CARE System will complete the system supporting the qualification process and: it will associate, with some human help, every component with the functional and interface specifications and will label it with a set of classification attributes similar to the one we have described in the last section. The prototypes will run on a Sun Workstation under the UNIX operating system. We are eventually planning to integrate the CARE System and the TAME System /20/, developed at the Computer Science Department of the University of Maryland.

7. References

/1/ V.R.Basili, H.D.Rombach, J.Bailey, B.G.Joo, "Software Reuse: A Framework", *Proceedings of the Tenth Minnowbrook Workshop on Software Reuse* , July 28–31, 1987.

/2/ SPC, *Proceedings of the Workshop on Software Reuse* , Technical Report SPC–TR–88–008, Software Productivity Consortium, Reston, VA, October 1987.

/3/ B.A.Burton, R.W.Aragon, S.A.Bailey, K.D.Koehler, L.A.Mayes, "The Reusable Software Library", *IEEE Software* , July 1987, pp.25–33.

/4/ T.Biggerstaff, C.Richter, "Reusability Framework, Assessment, and Directions", *IEEE Software* , March 1987, pp. 41–49.

/5/ R.Prieto–Diaz, P.Freeman, "Classifying Software for Reusability", *IEEE Software* , January 1987, pp.6–16.

/6/ E.Seidewitz, M.Stark, *General Object-Oriented Software Development* , Software Engineering Laboratory, NASA Goddard Space Flight Center, SEL–86–002, August 1986.

/7/ W.Tracz, "Ada Reusability Efforts: A Survey of the State of the Practice", *Proceedings of the Joint Ada Conference 1987* , March 16–19, 1987, pp 35–44.

/8/ G.Brooch, *Software Components with Ada* , Benjamin Cummings Publishing Company, 1987.

/9/ N.H.Madhavji,"Fragtypes:A Basis for Programming Environments", *IEEE Transactions on Software Engineering* , vol.14, no.1, January 1988, pp.85–95.

/10/ D.Hutchens,V.R.Basili, "System Structure Analysis: Clustering with Data Bindings", *IEEE Transactions on Software Engineering* , vol.11, no.8, August 1985, pp.749–757.

/11/ V.R.Basili, H.D.Rombach,J.Bailey, A.Delis, F.Farhat, "An Ada Reuse Metrics", *Ada Reusability and Metrics Workshop* , Atlanta, Georgia, June 15–16, 1988.

/12/ V.R.Basili, H.D.Rombach,"Software Reuse: A Framework", in preparation.

/13/ S.D.Conte, H.E.Dunsmore, V.Y.Shen, *Software Engineering – Metrics and Models* ,The Benjamin/Cummings Publishing Company, 1986.

/14/ V.R.Basili, H.D.Mills, "Understanding and Documenting Programs", *IEEE Transactions on Software Engineering,* Vol.SE-8, No.3, May 1982.

/15/ *Process Design Language/Ada,* IBM Federal Systems Division, 1983.

/16/ D.C.Luckham, F.W.Von Henke: "An Overview of ANNA, A Specification Language for Ada", *IEEE Software,* March 1985, pp. 9-24.

/17/ Joseph A.Goguen, "Reusing and Interconnecting Software Components", *IEEE Computer,* February 1986, pp.16-28.

/18/ R.Prieto-Diaz and J.Neighbors, "Module Interconnection Languages", *Journal of Systems and Software,* 6, pp. 307-334, 1986.

/19/ ISO TC97/SC5/WG3, *Concepts and Terminology for the Conceptual Schema* J.J. van Griethuysen (ed.), 1982.

/20/ V.R.Basili, H.D.Rombach, "Tailoring the Software Process to Project Goals and Environment", *Proceedings of the Ninth International Conference on Software Enginnering,* March 30-April 2, 1987, Monterey, CA, pp.345-357.

/21/ B.Choi, R.DeMillo, W.Du, R.Stansifer, "Observing Ada Software Components", *Proceedings of the Sixth Symposium on Empirical Foundations of Information and Software Sciences,* October 19-21, 1988, Atlanta, GA.

/22/ 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.