The Generic Reusable Component: an Approach to Reuse Hierarchical OO Designs

Massimo D'Alessandro, Pier Luigi Iachini, Alessandra Martelli
Intecs Sistemi S.p.A.
Via Fratti 14, Pisa, Italy

Abstract

Researchers in software reusability have long realized that reuse is most effective when applied as early as possible in the life cycle. Yet most experience to date has been obtained at the relatively low code level. Departing from the Hierarchical Object Oriented Design (HOOD) method, which is fast becoming a standard design method not only in the European space industry but also in other industrial environments, we have developed a framework for the specification of generic reusable components existing wholly at the architectural design level. All features of the component necessary for its effective use are stored with the component, including documentation, design rationale, formal specifications, reuse history and access rights.

Design-level genericity is introduced to widen the scope of applicability of the component. At design level a generic component is a partially defined object whose behaviour depends on the behaviour of parts that are not yet designed. It is shown how to integrate the design of such components into the full HOOD life-cycle method.

1. Introduction

Many definitions of software reuse have been given, starting from a narrow one: "Reuse is the reapplication of source code" to a broad one provided in [2] which states: "Software reuse is the reapplication of a variety of kinds of knowledge about one system to another similar system in order to reduce the effort of development and maintenance of that other system. This reused knowledge includes artifacts such as domain knowledge, development experience, design decisions, architectural structures, requirements, designs, code, documentation, and so forth."

The approach we are proposing is to reuse all possible results of the various steps of the development process of a reusable software component.

In order to successfully reach this final goal, different kinds of knowledge are to be provided as parts of the reusable component. They can be divided into three categories:

- *Reusable knowledge*, which includes software requirements, design, source code, executable code, test data and plans, documentation, etc.
- *Knowledge enabling the reuse* of a software component. Starting to reuse a component at implementation time may result in a very hard, limited and, perhaps, frustrating job; at this time in the development process, most of the implementation choices (architecture, data structures, algorithms) have been already made and finding a component exactly fitting the needs is very difficult. Reusable components tend to be very small and low level and the cost of reuse may not be justified. In order to reuse larger code components, it is necessary to provide knowledge about previous phases of the development process and approach reuse starting from those phases.
- *Knowledge providing additional information*, which is needed for a better understanding of a software component, both to verify if it satisfies the reuser requirements and to aid the reuser to integrate it into the system under development/maintenance, but also to modify it when necessary.

The approach we propose is based on the design phase: thus the enabling knowledge is the architectural design of a software component, performed with the HOOD method (Hierarchical Object-Oriented Design) [8].

A HOOD design can be viewed as a collection of components, where each component has a specification and an implementation associated with it. Usually, there is one root component (the System) which represents the actual software system to be developed.

Yet another advantage of having a reusable component composed of design and implementation is that a component is certified to be of "good quality" not only with respect to implementation concerns, but also to design. Designs can be fine-tuned, eliminating errors and inconsistencies, in the same ways as code through evolution and therefore benefit in the same way from reuse; thus the design itself is more likely to be error free and reliable, and it has been well documented in the literature how expensive design errors are.

0-8186-3130-9/93 $03.00

The work presented here focuses on the definition of a Generic Reusable Component model suitable for approaching reuse at design level.

2. The Reusable Component

This section will deal with a detailed analysis of the nature of a reusable component; HOOD is used as the associated architectural design method.

The definition of a reusable component can be introduced through its main characteristics.

Definition

A Reusable Component (RC) is a *certified, fully developed*, possibly *generic* software component, available for the integration into a software product under development.

Furthermore, a RC allows *composition with other components* and provides *alternatives in architectural /implementation* features.

Certified means that it satisfies some reusability and quality criteria.

Fully developed means that it has passed all the steps of the development process.

Generic means that the RC is abstract enough to be adapted to particular situations. Some choices could be recognized depending on the context of reuse (e.g. the actual type of the nodes in a graph) and the structure of the RC has to reflect the abstract nature of the implemented entity encapsulating some generic features as parameters.

Allowing composition with other components means that the RC may interact with external modules (e.g. DBMS, User Interface libraries, other RC's and also user developed modules). It allows the component to refer both to entities which are assumed to belong to the design environment which the component will be imported into, and to entities which are external to the system under development. Reusers will bind the external references of the component with their own entities or even add new objects to their environment (e.g. by reusing other components) in order to provide the services required in the RC definition.

Allowing alternatives in architectural/implementative features means that, in order to provide a higher level of flexibility, more than one design and/or implementation can be associated with the RC. This feature allows the RC to be seen as a *family of components* [5]. This represents an abstraction (e.g. an abstract data type) and provides a set of actual realizations of such an abstraction. Thus several designs with different provided interfaces are available; the reuser is allowed to choose the most suitable one to the requirements to be satisfied. It has been also introduced to group reusable components with the same functional behaviour. This should ease the process of identifying a component to be reused.

Because reuse is approached at software design level, an RC could be viewed as a family of HOOD designs and related requirements, plus the results of the various development process steps, like formal specifications, source code, linkable/executable code, test plans and test data, setting no limit to the level and/or granularity of a RC (it can vary from an entire system to a terminal component).

Note that there is an analogy between the RC structure and the phases of its life cycle. All the documentation concerning the definition of the problem which the RC represents (e.g. requirements analysis, system specification,...) is contained in the so called **RC Handle.**

Each design is one possible solution to the problem stated in the Handle, and it can be implemented in different ways in turn.

An RC Handle could also include a formal specification which allows component behaviour to be specified in an abstract manner, i.e. independent of implementation details. Thus, the Handle is more problem-oriented than implementation-oriented. This means that commonalities between the problem at hand (i.e. the system under design) and previously solved problems can be more easily identified and potential for reuse recognized. In fact, the name *handle* has been chosen since it is thought of as the means to access the RC during the searching phase of the Reuse process.

Figure 1 shows the structure of a RC as a family of different designs collected under the same handle.

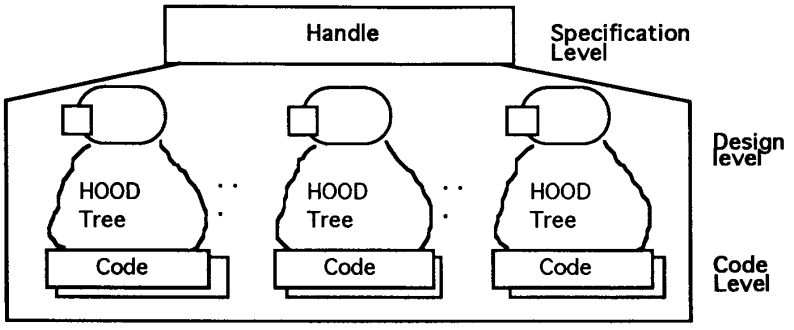

figure 1

3. The Generic Reusable Component (GRC)

A Generic Reusable Component is a partially defined RC, that is, an object whose behaviour depends on the behaviour of some not yet built parts. Such undefined parts are constrained to have certain properties so that the resulting component has a correct meaning. In this way a possibly infinite number of RC's are abstracted into one component and for this reason a GRC is also called "class". Some standard alternative instantiations can be provided in the library or left to the user. Class mechanisms are very suitable for reuse, since they allow factorization of sets of RC's and, on the other hand, they allow the user to personalize the class in a way not foreseeable by its builder by instantiating its undefined parts.

This section shows how the GRC model provides genericity and references to external modules. For the sake of simplicity let us assume that a GRC contains just one design. From a conceptual point of view, the link between a GRC and an RC is mainly provided by the fact that if a GRC does not present any generic features, it is an RC.

Since a GRC is mainly structured into two levels, Handle and Designs, the genericity appears at both levels.

3.1. Genericity

Genericity at Handle level means that the handle doesn't identify just a single problem but rather a *space* of problems. The GRC Handle expresses the genericity of the component, providing the following structure:

- the *parameter declarations*
  this is the list of the identifiers of the entities considered as parameters, together with their possible constraints,
- the *body description*
  this is the description of the semantics of the component in which the identifiers of the parameters must occur.

This definition is broad enough to support several choices of description languages, including the natural one. The choice of a particular specification language to be used in the description of a component is not a concern of this paper.

At design level a single design must allow the implementation of each problem belonging to the problem "space" of the GRC handle. This is partially achieved by the HOOD class parametric mechanism (very similar to the Ada generic concept). However, additional concepts have been added to make Reusable Components more flexible.

3.2. To Be Refined Objects

A new type of object has been defined to be used in a HOOD design as provider of generic features. Objects of such a type are called *To Be Refined Objects (TBR Objects)* since they provide services whose semantics are only partially defined, i.e. they are constrained to have "at least" some properties but are not completely specified.

Thus an object that uses a TBR Object is generic with respect to the used items. In other words, a TBR object looks like a formal parameter in the HOOD structure, while its specifications play the role of the constraints upon actual instantiations.

Obviously a TBR Object does not contain any refinement; it is just an object in which only the interfaces are given (provided interface, required interface), i.e. the signature of the operations, the specification of their constraints (formal or informal) and indication of which existing objects have to be used to implementing them.

Using TBR Objects in a HOOD Design allows it to become generic, while replacing them with real HOOD objects, which satisfy the TBR objects constraints, permits the creation of an actual instance of the design.

It should be noted that we have not modified the HOOD design method. A HOOD design with TBR objects is just a meta-HOOD structure (Architectural template). A suitable toolset can manage such a meta structure and produce a HOOD design to be imported by reusers into their own environment.

3.3. External Objects

Besides genericity, another important feature must be added to the GRC in order to achieve its composition with other modules when the component is reused. This feature consists of the possibility to refer to *External Objects*. The TBR objects concept covers this feature as well.

External objects have the same characteristics as TBR objects, but they are intended to model objects that belong to the importing environment of the component. Their interface specifies the operations that the actual object (the one in the design where the component will be reused) has to provide.

From a design point of view External Objects are TBR objects used by the root object of the GRC design. The reusable component providers can imagine that their component interacts with objects of the reuser's design under development by using External Objects as its partners. Whenever a reusable component is imported by reusers, its External Objects must be replaced with actual objects existing in the reusers' design.

3.4. Roles of design parameters

TBR objects (and External objects as well) are formal parameters of the HOOD structure and they can play different roles with respect to the GRC generic semantics:

- *Architectural parameters*
  they contain an essentially complete specification, or have a particular role in the design such that any consistent instantiation of them does not provoke modifications to the external behaviour. The provided

interface semantics of the root of the Design are not affected by their instantiation.

- *Semantic parameters*
  they have an incomplete specification or have a particular role in the design to allow choices affecting the overall semantics to be taken at instantiation time. The provided interface semantics of the component change depending on the instantiation of the TBR object.

In the first case there is no impact on the GRC semantics and the name "class" could be considered as improper. It is only a matter of different implementations of the same thing, i.e. the design is parametric only from an architectural point of view.

In the second case the semantics of the GRC is parametric and it represents a class of different projects (different requirements).

Besides TBR objects, Constant, Operations and Types can be parametrized in a GRC as well. They are not related to a particular object of the HOOD tree but they are directly supported by the HOOD class concept. They can also be classified as Semantic or Architectural, depending on their role in the design's functional semantics.

3.5. Traceability

Since genericity is expressed both at handle and design levels, a way to relate handle parameters to design parameters is needed. Thus, each handle parameter is in relation to a set of design *Semantic parameters*. This relationship is called *parameter traceability*, and allows reusers to understand how a handle parameter is implemented in a certain design by TBR objects. Since a GRC can contain several designs, there is a different relationship for each of them. Figure 2 shows the traceability relationships in a GRC.

Textual and possibly formal information (e.g. where a parameter is mapped to, what is its semantics, etc.) is also available on each link to detail the correspondence between the level of parameters.

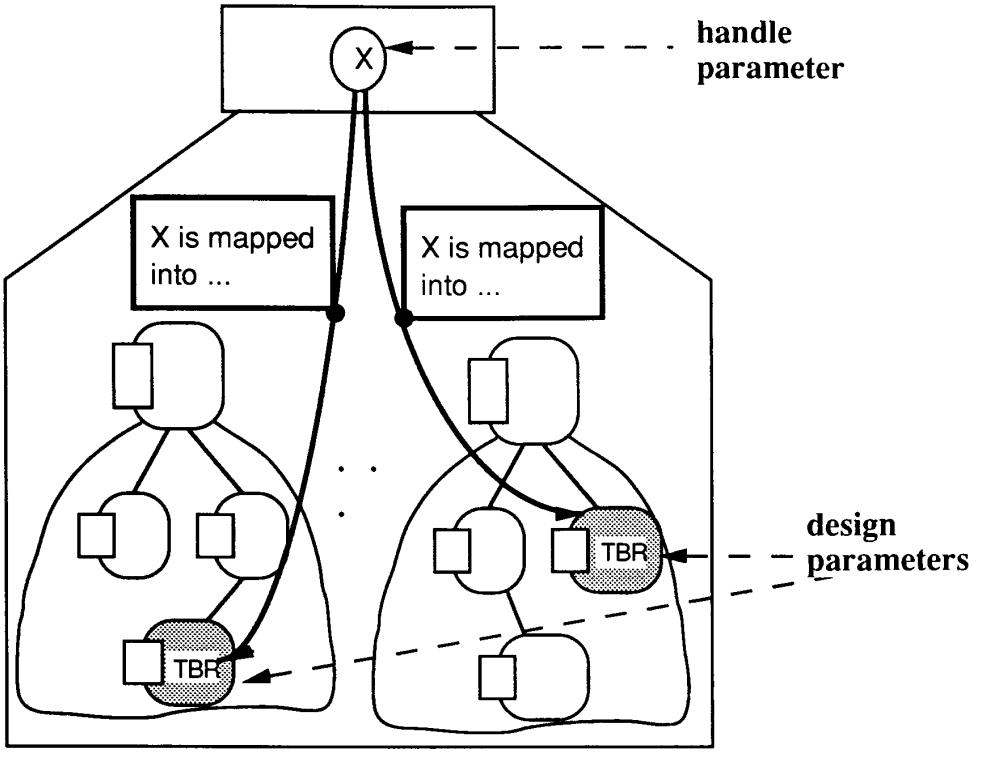

figure 2

4. Instantiation

As previously remarked, reusers have to work with HOOD objects in their environment. This section shows the process for obtaining a HOOD design from a GRC.

During the browsing of the library, a GRC is selected when the reusers discover that one of its possible instances is a useful reusable component. After selection, the instantiation process starts. It consists of several phases:

1. Instance Specification

The formal parameters in the GRC description are replaced with an actual specification. After this phase is

completed, the "actual problem" is defined but not yet implemented.

2. Choice of a Design

One of the available designs of the GRC is selected depending on the architectural requirements of the reuser.

3. Choice of an Implementation

One of the available implementation configurations associated with the chosen design must be selected as well.

*4. Definition of the Actual Parameters Specifications (**AOS**: Actual Object Specification) for semantic parameters*

This is the most crucial phase of the instantiation. The reusers have to define the specification of the actual objects which the design parameters will be replaced with. Such a definition must take into account the instance specification given in phase 1. That is, it has to be consistent with it so that the design results in an exact implementation of the instance. The traceability between the GRC abstract parameters and the semantic parametric objects of the design, documented in the design itself, allows the reusers to verify the correctness of the instantiation. Note that, at least from a conceptual point of view, both definition and implementation of the AOS's can take place without knowing the details of the architectural design of the GRC.

5. Possible choice of available implementations for architectural parameters

Since the GRC structure allows an architectural parameter to have several implementations stored in the library, the users can choose one of them instead of delaying the development to the integration phase.

At this point the HOOD design is completely defined. It has only one design and implementation and well defined semantics. The only deferred features are the implementation of the *Architectural parameters* and the implementation of the AOS specifications created in the instantiation process. Since such an implementation activity is nothing else than a standard HOOD development, it would not make sense outside the HOOD design environment of the reusers. This is the main reason why these implementations are delayed.

When reusing a GRC design, its *external objects* become the requirements for integration. In other words, they are still objects whose implementations are deferred but, instead of being developed inside the structure, they can be either "unified" to other objects belonging to the importing environment, or developed as any other required object.

In the case where formal specifications are used as the Handle description language, they could be exploited to support this process. In fact, critical applications may require a proof of consistency between phase 1 and 4, both in the definition of the AOS specifications and in the integration of the produced HOOD design.

5. Hierarchies of GRC's

Genericity is a good feature for Reusability. It allows commonalities between modules to be kept in one single place, i.e. the generic module. Unfortunately HOOD cannot capture fine grains of commonalities [9]. This is because they have only two levels of modules: generic modules and fully instantiated modules. Thus is not possible to describe a complex hierarchy of modules that have different levels of parametrization.

The possibility to specify only some of the GRC parameters is very attractive to achieve a more flexible reuse model. A GRC is said to be more specified than another one if the former has been obtained by instantiating a subset of the parameters of the latter (*instance_of* relationship). Thus each GRC in the library can be in a relationship both with a more general component (parent) and with more specific components (children).

The Partial Instantiation mechanism has several applications to component construction and reuse. First of all it can be employed to obtain new GRC's from the existing ones, providing a sort of *provider-oriented reuse*. The component provider can recognize a partial instance of an existing GRC as a suitable reusable abstraction; thus, instead of reimplementing it from scratch, the provider can just instantiate a subset of its parameters generating a new GRC. As a special case, a copy of an actual (complete) instance can be inserted in the GRC as a partial instance of its original GRC.

Furthermore, by structuring a GRC as a tree of partial instances, a *factorization* of information is made both at specification and design level. This leads to a more efficient storage and to a better understanding of the components, since common concepts and architectural choices are not replicated in the Reusable Component Library. This idea is similar to the concept of *program families* introduced in [6] as a future way of representing reusable components.

However, the availability of partial instances of a GRC is useful for the reusers to get an insight into its power as a component template. Looking at examples of instantiation of a generic component, the reusers can better understand how new instances can be obtained from it.

Finally, the *instance_of* relationship provides a further structure in the library in addition to the classification schema used in the retrieval operation. By navigating *instance_of* relationships the reusers can exploit such a structure to make retrieval more efficient and flexible.

As a consequence of the two-level structure of a GRC, the *instance_of* relationship exists both at Handle and Design level. Following the instantiation process described above, after a partial instantiation of a GRC Handle (instance specification phase) there must be a partial instantiation of its designs. In other words, let us assume that a generic description **handle** is instantiated to obtain **handle'**. Then a design **D'** for **handle'** can be obtained by instantiating a design **D** of **handle**. Thus **D'** is a partial instance of **D**, and they are in the relationsh *instance_of* to each other.

The *instance_of* relationship is defined at design level as well. Thus, if two designs are in an *instance_of* relationship, then their related Handles must also be in that relationshp; that is, the two levels of the relationship are strictly coordinated.

Furthermore, once a new handle has been obtained by partial instantiation, a design for it might be defined from scratch, following the specification. Hence, a handle obtained by partial instantiation can have two kinds of designs:

1) *derived designs*
   which are obtained by partially instantiating the objects of the original design;

2) *basic designs*
   which have been built from scratch.

Figure 3 shows an example of relationships between handles and designs. In the figure, **Handle2**, **Handle3** and **Handle4** denote partial instances of **Handle1**. **Handle2** has only derived designs, namely **D1'** and **D2'**, respectively obtained from **D1** and **D2** belonging to **Handle1**.

Handle4 has a derived design (**D3''**) and two basic designs (**D5**, **D6**), while **Handle** can only have basic designs, since it is the root of the hierarchy.

The situation shown in **Handle3**, in which the design **D4** is derived from the design **D3''** belonging to **Handle4**, cannot take place since **Handle3** and **Handle4** are not in an *instance_of* relationship.

Note that the navigation through the *instance_of* relationship is possible at design level as well.

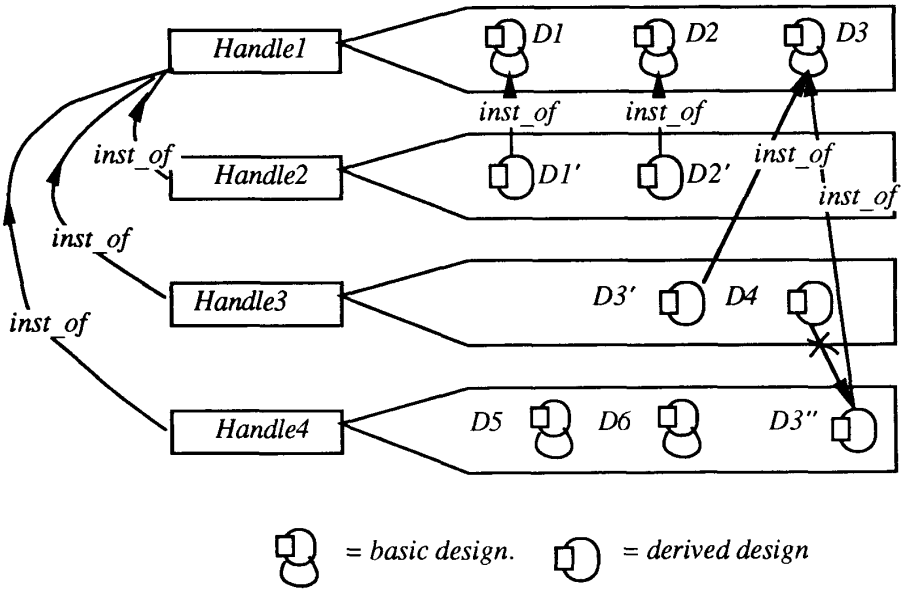

figure 3

6. Classification of GRC's

The given GRC model has an impact on the classification method used to allow retrieving of reusable components.

As a matter of terminology, the *classification model* is the philosophy of classification (e.g. hierarchical, faceted,.etc.), while the *classification scheme* is a particular realization of a classification according to a *classification model*. Thus, for example, Booch's component taxonomy is a *classification scheme* of the hierarchical *classification model*. [4]

First of all, the GRC is not just a monolithic component, but it includes in itself a family of subcomponents (different designs and code implementations), so the classification mechanisms are involved in the internals of the GRC as well. In other words, by collecting under the same Handle several equivalent designs, a first classification task is performed.

Instead of mixing Semantic (functional), Architectural (non-functional) and even Implementation features in the same classification scheme, the GRC model forces us to use a more structured approach.

There are four levels of classification in a library holding GRC's:

- the *Semantic classification* level.

This is the highest level, since it involves the Handles of the components. Depending on their semantics, the GRC's are classified in the available classification scheme. Correspondingly, the retrieval phase starts with the search for a suitable behaviour that is needed in the reuser project. That is, the reusers first

look for a GRC handle whose description matches their functional requirements.

It is worth noting that the more abstract the specifications are, the easier it is to understand whether the components under examination can be reused.

Note also that the choice of collecting equivalent designs through the Handles does not overload the classification task. In fact, the classification mechanism, in order to be effective, has to be quite general and cannot express too much detail, in order to collect only equivalent components in the same class.

On the other hand, since in the Handle we can use formal specifications, the semantics of the designs can be expressed at any level of detail [12].

• the *Architectural classification* level.

This level is concerned with a particular GRC Handle. Once the Handle is established, several designs are available for it inside the GRC. Each of them holds some architectural peculiarities that have to be classified as well.

As in the Booch taxonomy, each design is a different *form* of the component: there may be a parallel and a sequential version, a memory managed version, and so on.

Furthermore, since the universe of concepts involved in this level is different from the previous levels, a separate classification scheme should be provided to classify the different designs of a GRC.

• the *Code Implementation classification* level.

This is analogous to the Architectural level. However the universe of concepts involved may be different so that even another classification scheme may be needed.

Some classification criteria should be defined to aid the reuser when he/she has to choose an available implementation associated with an architectural parameter. However the problem is analogous to classification at the Architectural level.

Thus the GRC allows different classification schemes, and even classification models, to be used at different abstraction levels. Figure 4 summarizes the above remarks.

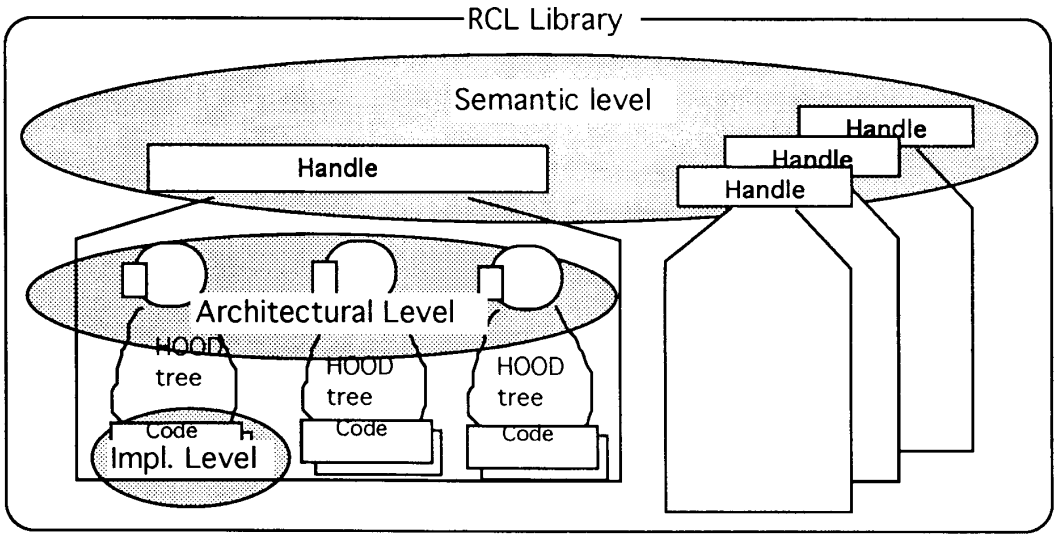

figure 4

7. Conclusions and future work

In this paper advantages of Reuse starting from the early phases of the software life cycle have been discussed. Design level has been shown the most suitable framework to deal with Reuse.

A Reusable Component Model has been defined to be as flexible as possible to maximize its reusability. Genericity and multiple architecture have been chosen as means to implement such flexibility. Starting from genericity at the specification level, a method to represent it in a HOOD design has been shown through the concept of *objects to be refined* (TBR objects).

Moreover, a method to obtain HOOD designs from a GRC has been described (instantiation method). It starts from the specification of the component (GRC Handle) down to the implementation level. Some investigations could be made into formalizing this procedure, in order to use it also in software developments where module correctness needs to be formally verified.

A tool supporting the component model and the instantiation method has been prototyped: RobinHOOD (Reuse Objects in HOOD). RobinHOOD provides functionalities to create, store, retrieve, browse and instantiate GRC's. Some practical applications of the RobinHOOD reuse model are needed to assess the real usefulness of the defined concepts. Thus, some case

studies on the prototype will be necessary to obtain feedback to the RobinHOOD specifications.

Acknowledgements

Some of the results reported in the present paper were originally elaborated in the RACE 1021 ARISE project. The authors gratefully acknowledge John Favaro for his valuable comments and help.

References

[1] Biggerstaff T.,Richter C., "Reusability Framework, Assessment and Directions" in IEEE Software, vol. 4 No.2, Mar 1987, pp.41-49.

[2] Biggerstaff T., Perlis A., "Software Reusability", acm PRESS, 1989.

[3] Boehm B.W., "Software Engineering Economics", Prentice Hall, 1981.

[4] Booch G., "Software Components with Ada", The Benjamin/Cummings Publishing Inc., 1987.

[5] M. Dausmann, "Organizing Component Libraries", AdaEurope Software Seminar, June 1988.

[6] W.B.Frakes and P.B.Gandel, "Representing Reusable Component", information and software technology, vol 32 no 10, december 90.

[7] Hoare C.A.R.,"Communicating Sequential Processes" Prentice Hall, 1985.

[8] HOOD Working Group, European Space Agency. *HOOD Reference Manual,* wme/89-173/JB edition, September 1989. Issue 3.0.

[9] B. Meyer, *Reusability: the case for O.O. design,* in "Software Reusability" Vol.2, ed. by T.Biggerstaff and A.Perlis, ACM press, 1989.

[10] R. Prieto-Diaz, *Classification of Reusable Modules,* in "Software Reusability" Vol.2, ed. by T.Biggerstaff and A.Perlis, ACM press, 1989.

[11] Tracz W.,"Software Reuse Myths" in ACM SIGSOFT Software Engineering Notes, vol.13 No.1, Jan 1988, pp. 17-21.

[12] R.di Giovanni, P.L. Iachini, "HOOD and Z for the Development of Complex Software Systems", LNCS 428, Springer Verlag, April 1990.

[13] Wegner P., "Capital-Intensive Software Technology" in IEEE Software, vol. 1 No. 3, Jul 1984, pp. 7-45.