Reusing Requirements through a Modeling and Composition Support Tool

M.G. Fugini §† *M. Guggino* † *B. Pernici* ‡†

§Università di Brescia
‡Università di Udine
†Politecnico di Milano
piazza Leonardo da Vinci 32, Milano, Italy
relett14@imipoli.bitnet

Abstract

This paper presents the concepts and tools for reusing requirements being designed and implemented within the ITHACA project. [1] The RECAST (REquirements Collection And Specification Tool) tool guides the Application Developer in the requirement specification process by providing suggestions to the reuse of components. To this aim, RECAST includes a meta-level of definitions; here, meta-level classes associated to components contain design suggestions about the reuse of these components and about the design actions to be performed during the subsequent application development phases.
A uniform approach is provided in RECAST for specifying both components and meta-level classes: the ORM specification model, especially oriented to requirement specification of object-oriented applications. The paper describes the components connection model, the environment for reusing requirements, and the interface of RECAST.

1 Introduction

The purpose of the ITHACA Application Development Environment (ADE) /Ader 90/ is to support application development through reuse of development information regarding both available executable software and development information, such as requirements, scripts, design documents, design decisions and motivations. To enhance the effectiveness of this ADE, an object- oriented approach to application development /Cox 87, Shlaer 88/ is considered in ITHACA, and a centralized repository of development information and reusable components is the core of the environment. This repository, called Software Information Base (SIB), contains descriptors of development information /Constantopoulos 89/. It is accessed by the ADE tools in order to retrieve reusable elements and progressively build an application /Gibbs 90/. In particular, one tool is devoted to the requirement specification phase of ITHACA; the RECAST tool /Fugini 90a/ guides the Application Developer (AD) in the exploration of the characteristics of the application at hand and in the selection of useful requirement components. This tool is based on design knowledge stored in the SIB in the form of metaclasses,

[1]This work is partially supported by the ITHACA Esprit II Project n. 2705

that is, of meta-level descriptions associated to reusable classes which detail the usage, interconnection modes, the necessary interfaces, the optional design choices that can be undertaken by the AD to appropriately reuse the available **components**. Classes and meta-level classes considered by RECAST are expressed in a homogeneous formalism using the Objects with Roles Model (ORM) /Pernici 90a/, explicitely defined in ITHACA for designing object-oriented systems under a reusability approach. ORM allows one to specify the structure and the dynamics of object systems through the concepts of roles, states associated to these roles, messages, and a rule-based mechanism for object evolution. In particular, for meta-level classes, these ORM concepts are used to provide the AD with design suggestions about how a component can be selected, tailored, modified, adjusted to the requirements of the current application.
This paper presents the current work being performed on the RECAST tool. First, the definition of components, as elements that can be selected from the SIB, examined in a temporary workspace, interconnected and tried by the AD and then put together upon a design "committment" is given. An interconnection model is outlined, with special attention to the graphical representation. The graphical representation considers a notation for components which are **selected explicitely** by the AD, a notation for components which are **proposed** by RECAST as necessary or optional components to be consequently selected, a notation for connections between components and a way of representing various **detail levels** during the specification process. Then, the paper illustrates the concepts of **framework**, as a set of reusable classes which are related in a reusable schema. In ITHACA the concept of framework is related to application domains; examples in the domain of Public Administration Offices /Kappel 89/ is given. From the requirements viewpoint, a framework is a set of components and of their associated meta-level classes which are linked together to give the AD the guidelines necessary to reuse the components. The use of frameworks in a specification method is discussed.
An important aspect of RECAST is its **user interface**. The problem of orientation in the development information space quickly arises when large class collections are to be explored /Gibbs 90/. Moreover, the architecture of RECAST has to take into account carefully the interaction with the SIB, with the Selection Tools defined for the SIB /Costantopoulos 89, Fugini 90c/, and the interaction with the VISTA design tool /Stadelmann 90/, which provides support to the detailed design (scripting) phase of the ITHACA development process via a graphical interconnection interface.
The paper is organized as follows. Section 2 describes the concept of component and the operative model for interconnection of components. The ORM model used for components is briefly reviewed, and the graphical notation used for components and components interconnection is described. Section 3 illustrates the environment for reusing requirements based on the concepts of framework and of meta-level classes, as concepts which incorporate design guidelines to the AD. Section 4 describes how semi- automatic support to requirements reuse can be provided: the current design of RECAST, and in particular of its interface to the AD and to the ITHACA ADE tools, is given.

2 Requirements Modeling and Composition

In this section the Object with Roles Model is briefly presented and a graphical representation of ORM classes suitable for the requirement composition process is presented. Central to the Object with Roles model is the concept of **role**. A role is a **context** for message interpretation, and corresponds to an object's specific behavior during its execution. The role concept relaxes a restriction that class models impose to an object, that is the inability of an object to exhibit more than one behavior at the same time. ORM objects can play multiple roles at the same time, for example a person can be driving a car, listening to the radio, work for a company, be a husband or a wife during its lifetime, with many of this "roles" possibly played at the same time. ORM objects intrinsically exhibit a form of concurrence and communicate exchanging messages. Roles can be created and manipulated during object execution.

2.1 ORM Modeling Concepts

ORM Classes are defined as an ordered pair consisting of a unique class name and a set R of roles:

class = (cn, R)
R{R0, R1,..., Rv}

Each class describes different roles (which correspond to different behaviors). Any role can be played independently (subject to some constraints to be discussed later) and an object is allowed to play many role instances at the same time, even multiple role instances of the same role type.
The global characteristics of an object are described by the Base Role type R0. Each role type is specifies as:

Ri = (rni, Pi, Si, Oi, Rui)

where:

- rni is the role's name;
- Properties Pi are abstract description of data and are implemented by instance variables. Each Pi has a name ni and a domain di. For example, a Person class might have (name, string) as a property-value pair.
- States $s - i \in Si$ describe the abstract role state for a given role.The object complex abstract state is defined as:

  state = (s-i0, s-i1, s-i2,....,s-in)

  where s-jk is the k-th instance of j-th role.
- Operations Oi are the set of messages an object can send or receive. Messages are indicated by a name and are preceded by an $\leftarrow$ if incoming, by an $\rightarrow$ if outgoing.
- Rules $r - ui \in$ Rui define the role's behavior, stating which messages can be sent and received in each state. To each message sent or received, a state transition is

associated. Only messages that are defined in association with a state are allowed to be sent and received in that state. State transition diagrams show the evolution of objects according to state transition rules.

The base role, of role type r0, is always instantiated exactly once, and is responsible for the other roles' creation, destruction and coordination. New roles can also be instantiated by external add-role messages. The reader interested in a more detailed discussion of the ORM model is referred to /Pernici 90a/.
A sample ORM class definition is given in the following; where the person/office class is defined, with five roles, R0, ..., R5. R0 and R1 are detailed in their properties, states, messages and rules.

```
class= (person/office,
        R0=  <base-role,
            properties = { (name, string), (id, string)},
            states = {active, suspended},
            messages = {<-add-role, <-suspend-role,
                        <-resume-role, <-terminate-role,
                        <-suspend-object,  <-resume-object,
                        <-kill},
            rules = {  } >,

        R1 = < requester-for-approval,
             properties = {(ref-doc, document)},
             states = {started, waiting, completed},
             messages = {->request, <-approve, <-reject},
             rules =
                {rule-req,0: msg(<-add-role) => state(started)
                                     /*initial role  state*/
                 rule-req,1: state(started), msg(-> request)
                                        => state(waiting)
                 rule-req,2: state(started), msg(<-approve)
                                        => state(completed)
                 rule-req,4:
                      constraint(state(waiting), msg(<-approve)
                                  =>forbidden-msg(<-reject)))
                 rule-req,5:
                      constraint(state(waiting), msg(<-reject)
                       => forbiddenmsg(<-approve)))
                       }>,
      R2 = <request-handler, ... >
      R3 = < reminder-informer, ... >
      R4 = <document-preparer, ... >
      R5 = <approver, ... >          )
```

2.2 Components

In this section and in the following, the "operative model" of requirement composition is described. Since the trend within the ITHACA project is to have graphical tools and visual representation, a graphical representation of the composition process is also described, briefly mentioning the most important phases where the tools support the AD in his process of requirement composition. ORM classes are graphically represented as **rectangles**, and are shown as partitioned in several **slots**, each corresponding to a role of the class. ORM classes interact playing roles that communicate exchanging messages. The point where the interaction between roles of the same or of a different class is considered to take place is called *role pin*. This is graphically represented as an overhanging element from the class in correspondence of the role's slot.

Requirements *components* are ORM classes or class roles. These are combined by the AD into a requirement specification, by selection from the SIB. The guidelines for the reusability of components are given in the form of ORM meta-level classes /Pernici 90b/ that are also stored in the SIB and whose roles and properties describe the design actions associated to a given component. Meta-level classes and design guidelines will be illustrated in more detail in Section 3. In our example, let us suppose that the role requester-for-approval of the person/office class be selected by the AD from the SIB as starting component for the requirement composition process. Some other components should or might be selected from the SIB as well, as a consequence of the AD's choice. In Figure 1, the person/office component is shown using the graphical representation described here above of classes and roles. In the figure, the state diagram associated to the base role of person/office is shown. This diagram depicts the state evolution of the class and can be zoomed in by the AD for examining the details of the class evolution in the given role. By zooming the state diagram of the base role of person/office, it is possible to see in detail the properties and rules of an ORM class. The suggested components to be selected upon the AD choice of using the person/office class are shown in Figure 1 with dashed lines. For each role-slot, the name of the role and the set of messages belonging to the role's Operation set are shown. The role's instance variables and rules (or alternatively the role's state transition diagram) may also be shown upon request, either temporarily (pop-up) or permanently. As a default, instance variables are shown only if they are actually declared in the component, i.e., the IsA hierarchy is not flattened. The role's state-transition diagram are shown in the usual notation /Harel 87/, and may be edited to modify the role's behavior. In particular, role diagrams are depicted using circles denoting states and arcs tagged with arcs tagged with the name of the message that triggers that transition and an optional message to be broadcasted to trigger other transitions in the role or in the connected roles. A slightly different visual representation of role state-diagrams allowing visual representation of messages will be discussed later on.

In Figure 1, the role requester-for-approval is shown with a continuous line because its has been selected explicitely by the AD; the Base Role is always selected as a consequence of any first role choice for a class.

The role document-preparer is considered *necessary* in the *design guidelines* for the

role requester-for-approval to operate properly, and its outline is shown with a dashed line, to show that its appearance is a consequence of the AD's decision, and not the result of the AD's direct inclusion of the role components. Since a necessary interface to the role document-preparer is considered to be the ORM document class, this class is automatically retrieved from the SIB as an implication of the person/office selection; the interaction of the two classes is shown with a role-link (to be discussed later) that connects the roles being-prepared of the document class and the role document-preparer of the person/office class (see Figure 1). All these components and their connections are automatically proposed to the AD as necessary components, and are depicted with dashed lines to show their being an indirect result of the AD's design actions.

2.3 Components Interaction

The ability of combining reusable components together in different ways to form different application is essential to the achievement of software reusability. As it has been pointed out /Tsichritzis 90, Ceri 90/, in most object oriented systems, classes are known to all objects, and this enhancement in reusability comes to the expense of imposed behavior uniformity. Hierarchical decomposition methods are in widespread use /Coad 90/, and Hierarchical Object Oriented Design methodologies are emerging /Hood 87/. In the following, a component composition method is proposed which basically supports hierarchical composition of components, thus encapsulating the innermost components behavior in a higher level abstraction.
Components are connected together using the concept of role-link. A role-link between role A1 of component A and role B1 of component B is a mapping of outgoing messages of role A1 into incoming messages of role B1 and viceversa. This mapping is considered to establish a message flow between the two roles, that is, it physically acts as a general message translation mechanism. The graphical representation of a role-link is a cable-like entity that connects two role-pins.

A role-link is depicted in Figure 1 between the document- preparer role of person/office and document; this role-link splits to the three roles of document. When a mapping exists between two roles which does not require message translation or elaboration (i.e., the message names remain the same), the roles are said to be **mutually compatible.** Roles should be checked for compatibility before a connection is made between them. In some cases, however, a broader interconnection capability may be necessary than that provided by the compatibility property, and should be supported. Role compatibility should be regarded as a sufficient, but not necessary property of connectable roles. Another basic concept for composition is the Process Class. A **Process Class** is an ORM class whose roles represent **tasks** of the application; process classes are used in the requirement composition process to coordinate the various selected components to model the whole behavior of the application under development /De Antonellis 90/.

The order of execution of these tasks can be modified by the AD by examining the global state-transition diagram of the process class and choosing between the various possible control sequences. In our example, an ORM process class is the Public Administration

Figure 1.
Selection of the person/office class
and suggested components.

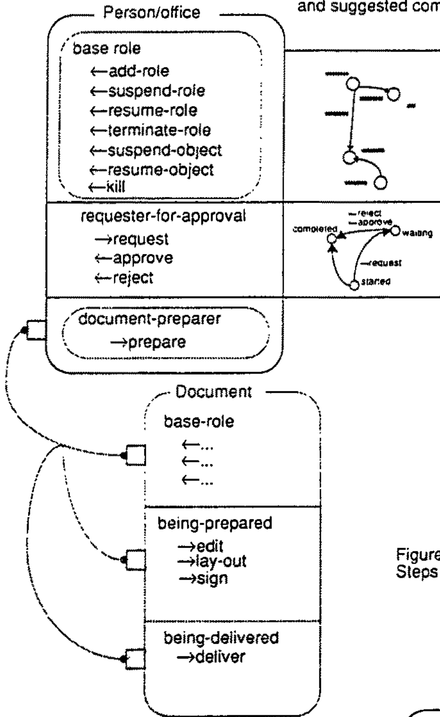
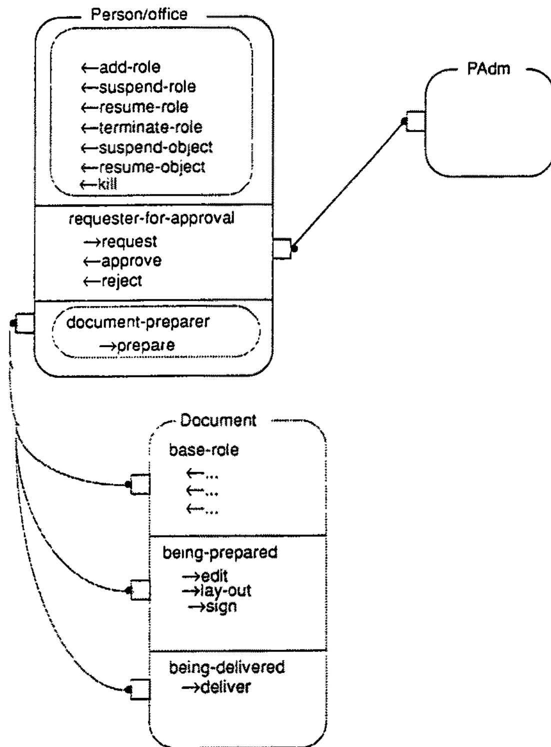


Figure 2
Steps of component connection

Figure 2a
Selection of PAdm



Office class. Its specification and that of some of the other classes of our example is given in beneath (we use a simplified notation).

```
process class PAdm
/* roles */
   base-role :
             properties = (name, string)
             states = { active, suspended, terminated}
             messages= {add-role, suspend-role,  resume-role,
                        terminate-role, suspend-object,
                        resume-object}
  internal :
              ...
  external:
               ...
  communication:
          ...


class Document
/* roles */
   base-role :
             properties = (name, string)
             states = {archived,being-processed, completed}
             messages= {add-role,  suspend-role,  resume-role,
                terminate-role, suspend-object, resume-object}
   being-prepared:
             messages = {edit, layout, sign }
   being-delivered:
             messages = {deliver}
   signature-handler:
             messages = {sign}
```

In Figure 2, the component composition steps of the PAdm example are represented. After selecting person/office, and obtaining document as shown in Figure 1, the composition proceeds selecting from the SIB the PAdm process class (see Figure 2a). As a default, upon selection the Base Role of the PAdm class is shown, with one role-pin that allows the AD to initiate the connection process. Connecting the requester-for-approval role with the PAdm component with a role-link will initiate the retrieval from the SIB of the roles whose presence is necessary in the PAdm Class in order to make the connection with the office/person class effective. The roles of PAdm are shown in Figure 2b; all roles are selected, and the visual conventions of Figure 1 apply.
Another ORM class in Figure 1 is the external-office; its interaction with the PAdm

Figure 2b
Details of PAdm component

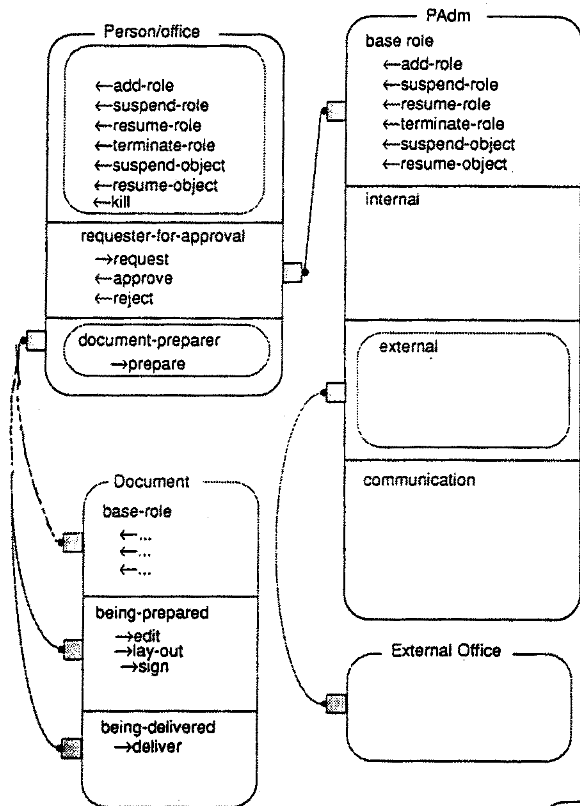
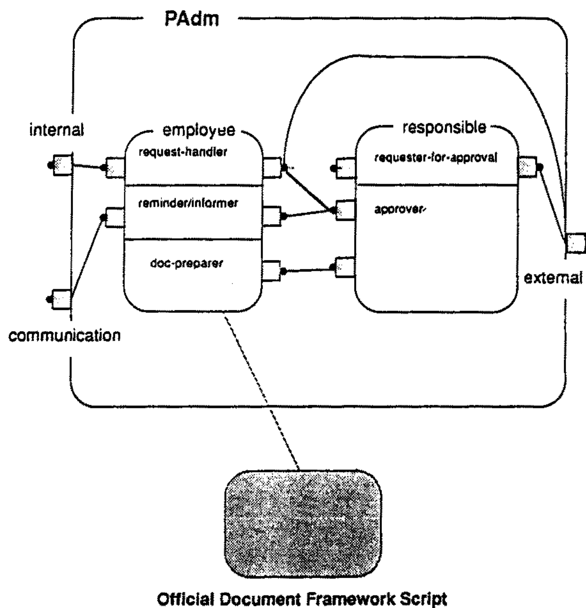


Figure 3.
Components of PAdm
Process Class



class is shown via a role-link. Whenever a connection is made from the AD choosing between different options, the design suggestions implied by that connection (i.e., the new role's necessary and optional interfaces) are shown and the other connection options are discarded.
Repeating the connection process, ORM classes may be graphically connected into complex aggregates until the desired specifications are composed. In the ITHACA terminology, this process of composition, with associated graphical notations, applies to all phases of the software process, and, at the design level is called **scripting** /Nierstrasz 90/.

ORM classes may themselves be composed of lower level classes as shown in Figure 3, where the inner components and structure of the PAdm class is detailed.
Corresponding to a deeper level of abstraction, the *inner workings* of the PAdm class are now shown. Two interacting ORM classes provide the external PAdm's behavior: *employee* and *responsible*. Both classes are specializations of the person/office class, and some of their roles, through a message translation mechanism, provide the required external role behavior. They correspond to the real-world counterparts "clerical worker" and "responsible" respectively.
Figure 3 depicts the interaction between the two and the outside world. The employee acts as a request-handler, a document-preparer, and a reminder-informer, corresponding to the roles of handling communications with the applicant and of performing office tasks internal to PAdm. He may also interact with an external office in its role of request-handler.

The responsible class does not interact with the request applicant (because the clerical worker act as a front-end in those roles); rather, it acts as an approver with respect to employee, and possibly reroutes requests for approval to an external office class (which in Figure 2b is represented as a consequence of choosing the optional role external in the PAdm class). A "related-to" link to the *framework script* official document can be followed when it is necessary to specify an official document structure. This type of connection brings the AD to another definition environment, called **framework** which will be illustrated in more detail in Section 3.
Here, we say that a framework script is a partially composed requirement specification that the AD can complete according to his needs. A framework script can also specify how external tools can be used to accomplish some steps of the specification process.
State transition diagrams visually illustrate a role dynamic behavior and are in widespread use and useful extensions have been devised /Harel 88/. For our purposes, they do not convey any important information when more roles are considered as connected and their dynamic behavior is to be shown. In fact, in conventional state diagrams, a state transition triggers a message ejection. Of these messages though, and of their flow, no visual indication is provided, usually considering them as broadcasted to a whole suitable *context*. An alternative graphical representation of state-transition diagrams is proposed here that, using the concept of *transition inhibition* and excitation, allows a visual representation of the message flow.

To represent the *flowing of a message* between two connected roles, a cable-like entity is

used, called a message-path. Message paths are considered to transport messages and to provide a visual indication of a message origin and destination. Message- paths may be tagged with the name of the message they transport and start with a component, called a **sensor** /Nierstrasz 90/ whose activation generates a message, that can either have a name and correspond to a role operation or be an internal, private message. Sensors are placed in correspondence of states, state- transition arcs or of message-paths, and are respectively activated by the system being in that state; a state-transition takes place on that arc or a message passing through the neighbouring message-path. When reaching the end of the message- path, the message reaches a component call excitator, which is in turn placed in proximity of a state-transition arc. When reached by the message the **excitator** will trigger the state transition. When more than one message-path flow to the same direction they can be aggregated and be drawn as a single entity, to be separated again when needed. This is mainly to avoid cluttering the diagrams and to provide visual indication of message groups. It is necessary in this case to provide information, either in the form of message names or using color and other visual clues, about how the message-paths exactly split again.
Each excitator along the arc must be active for the transition to occur; when more than one message can trigger a transition all their message paths reach the same excitator. When more than one message-path originates from a single sensor, that means that only one of the messages is generated. A particular auto-sensing configuration, in which a state transition is triggered by the state itself from which the transition takes place, is used to indicate that the system autonomously changes state without any external (represented) intervention.

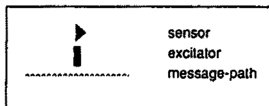
An example is presented in Figure 4 depicting the interaction of the role request-handler of Employee with the Responsible class.
In the example, *default* states are marked with a double circle, and the interaction between the roles of the classes is shown with its associated message flow. The interaction, which is initiated by the request-handler issueing a request, moves the responsible from the "available" to the "evaluating" state, whose possible results are a rejection or an approval. They both will move the request-handler from the waiting state to the completed state, thus terminating the role interaction.

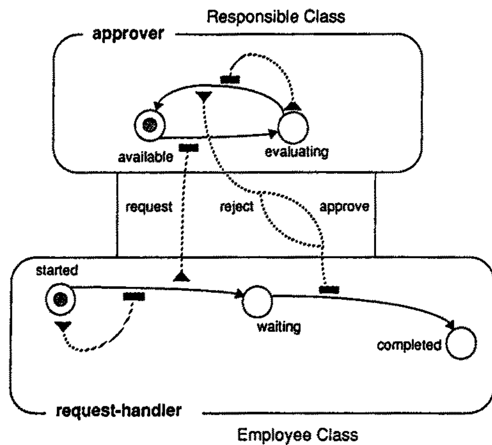
3 The Environment for Reusable Specification Components

In this section, we illustrate how the AD is guided by RECAST in composing an application specification. The environment guiding the AD is based on the concept of **framework**, as a set of related components, and on the **meta-level classes**, which incorporate design suggestions. The meta-level describes:

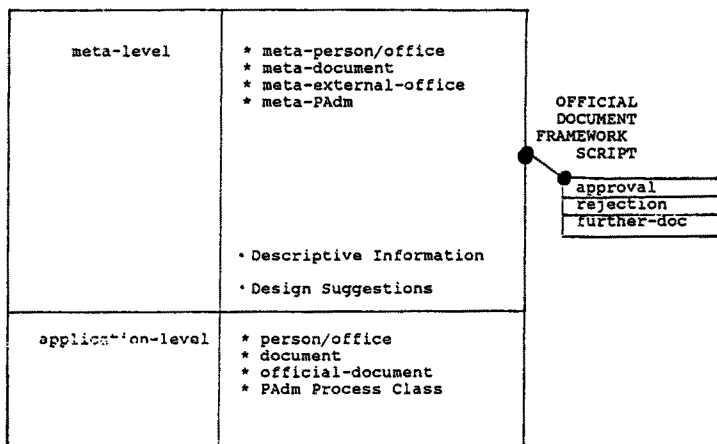
- how the components can be **reused**, that is, can be selected, refined, modified for being tailored to the application at hand;

Fig. 4
Roles interaction at the level of state diagrams.





PADM FRAMEWORK



5a) Structure of the Public Administration Framework

Fig. 5 Framework and metaclasses

- what **design actions** could or should be performed by the AD during the various moments of the design process;
- the **implications** of the design actions performed by the AD, for example, of the selection of components, of the definition of new components, of the specialization of existing components. Implications of design actions suggest to the AD the proper interconnections among components that should be set, according to the interconnection model of Sect. 2.

Moreover, meta-level classes are the mechanism by which links are established between the specifications and the subsequent phases of the development process.
The following of this section illustrates the framework concept and the idea of meta-level classes as elements containing design suggestions and therefore driving the AD in the development (Sect. 3.1); then, the use of these concepts in the specification process is described (Sect. 3.2).

3.1 Guiding the AD in the specification phase: framework and meta-level classes

The basis of the RECAST approach is that the AD is guided in finding in the SIB information relevant to develop a given application. The concept of *Generic Application Frame (GAF)* has been proposed within ITHACA /Nierstrasz 89/ to provide information to support the AD during the development process. The GAF for a given application domain should drive the AD to build a Specific Application Frame (SAF) for his application /Gibbs 90/. In the requirement specification phase of ITHACA, a GAF, or simply a framework, is defined as a set of classes related to a given application domain. Classes are organized at two levels /Pernici 90b/:

- the *application level*, describing the characteristics of objects belonging to that class;
- the *meta-level*, describing how the classes can be reused, that is, describing the design actions associated to the application classes.

The ORM model is used both for classes at the application level and for classes at the meta-level, the goal being that of having a *homogeneous object-oriented approach* to the development process /Pernici 90a/. Now, a framework groups the available classes and the meta-level classes containing the guidelines for class reuse within a given *application context*. It allows the AD to consider for reusability groups of components, rather than single components. According to the current approach in ITHACA, we assume that frameworks are established by application domains (banking applications domain, financial domain, public administration domain, and so on).
Frameworks relating to one domain can be regarded as constituting a semantic network of classes and meta-level classes where:

- *describes-reuse-of* links relate the meta and the application level;

- the *meta-level classes* constitute a *network of design knowledge* used for reusing components;
- the *application level classes* are the network of *reusable components*; their interconnections to compose an application are described in the meta-level. Within a framework, an inheritance lattice is defined on the classes, defining specific inheritance constructs based on the ORM concepts of class and role /Parmigiani 91/. Moreover, the framework describes dependencies between application classes. The following dependencies are considered:

  - classes components
  - classes which provide needed services to a given class
  - classes which provide potentially useful services to a given class
  - alternatives in the specification of a given class, with preferred or default choices.

These mechanisms are mainly based on a *retrieval system* based on names associated to frameworks, and to classes, roles and messages. Names assume therefore a particular relevance in the definition of a class. The SIB retrieval mechanism defined in ITHACA, comprising browsing and querying tools /Costantopoulos 89/, is used for retrieval of specification components.
In particular, frameworks can be accessed either by navigating in the SIB or by formulating queries based on keywords associated to the frameworks. The classification schema is in the style of /Prieto-Diaz 87/ and is based on a set of predefined characteristics of components /Fugini 90c/. Basically, *component retrieval* can occur on the names of roles and messages of ORM classes, and using additional keywords included among the properties of ORM classes. These keywords should be under control of the Application Engineer (AE) who is responsible for maintaining the consistency and operability of the whole ITHACA development environment /Nierstrasz 89/.
The notion of *similarity of components* and of correspondence between components at the various stages of the development life-cycle is also supported in ITHACA /Fugini 90, Petra 90/. It is beyond the purpose of this paper to illustrate the ITHACA Selection Tools; some keywords useful for the selection of ORM application classes are assumed to be defined in the components of our examples.

Meta-level classes are defined using the ORM model. The definition of their roles, called *meta-level roles*, includes the following properties:

- *application domain*: the set of application domains for which the described application class is intended;
- *level*: a set of levels can be defined for classes, depending on the various design abstraction levels where the AD is supposed to operate and therefore on the number of hierarchical decompositions which exist for a given class. As an example, the PAdm process class can be regarded as a whole with a name and a set of roles; at a lower abstraction level, e.g., at a deeper level of decomposition, the AD will

want to see the components of PAdm, and in that case he will be presented with the "employee" and "responsible" subclasses of "person/office", as described in Sect. 2. This level of detail allows the AD to examine which components provide the external behavior of PAdm observed at the previous level of abstraction. A possible sample set of levels for classes is the following:

$$\{\text{meta, top-level, } \{\text{components}\}\ \}$$

where meta means that a meta-level class exists for that component, top-level is the highest level of detail where only the base role is available, and components is a recursive set of hierarchical decompositions of the class. A sample top- level specification of the application class PAdm is given in the Appendix.

- *keywords*: the set of keywords describing the application class and its functionalities. These are useful for class retrieval, as described above. Moreover, keywords are associated to the meta-level role itself, in order to allow its retrieval within the meta-level class.
- *dependencies*: properties defining relationships among classes.
  The basic kinds of dependencies which are defined are:

  1. *Required Interfaces*: the classes needed by a given class to work correctly /Hood 87/;
  2. *Component Classes*: the set of classes which compose the given class;
  3. *Acquaintances*: the set of classes related to a given class.

- *application level roles*: the set of roles and the global role state diagram of the application class.
- *design suggestions*: information useful for class reuse; design suggestions are included in the rule part of the meta-role. This item will be illustrated in more detail in the next section.

Using frameworks and meta-level classes, the *requirement specification phase* in ITHACA is performed by the AD on the basis of design information provided in the framework. In particular, the specification is obtained by selecting one framework, exploring and selecting the application level reusable classes of the framework, guided by the metaclass definitions, by tailoring these definitions to the needs of the application at hand, and by composing the definitions through the components interaction model described in Sect. 2.
These steps do not necessarily occur in the sequence indicated here, and can be iterated several times.
The sample framework considered here is the Public Administration framework of the ITHACA workbench. We consider a request forwarded by private organizations, and authorizations released by the PAdm office, according to what illustrated in Sect. 2. The PAdm framework include the person/office, document, official- document, external-office, and the PAdm classes. The ORM definition of some of these classes has been

Fig. 5b) Sample Definition of Document Metaclass

```
meta     document

  /* meta-level roles  */

   R0  meta-functionality

        properties /* descriptive information */

             domain: (PAdm, Office_Applications)
             level : (meta, top-level)
             required interface: person-office
             components: header, body
             appl-level-roles:  (base, being-prepared,
                                       being-delivered)
             implementation suggestions: (graphical-editor,
                                       4GL,DBMS, scanner)

        messages

             instantiate-role  meta-signature
             select-role       (base, being-prepared)

        rules

             if select-role person/office.document-preparer
                 then
                     N select-role (being-prepared,
                                    being-delivered)

   R1  meta-signature

        messages
             select-role signature-handler

        rules
            N define-role signature-handler

   R2  meta-presentation

        messages define-header
                 define-colors
                 enter-examples
                 define-interface
                 define-official_document

        rules if define-official_document
                 then P select-framework OFFICIAL_DOC_SCRIPT
```

given in Sect. 2: those definitions regard the application level; a metaclass is associated to each of those classes, as illustrated in Figure 5a.

Reusable application classes and their corresponding meta-level classes compose the *PAdm framework*. In particular, to the meta-level of "document", a link is associated connecting the framework to an existing script containing information guiding the AD in the preparation of "official- document" classes (a specialization of the "document" class). As shown in Figure 5a, the "official-document" can in turn have the roles (approval, rejection, further-documentation) corresponding to the result of the request processing activities performed by the PAdm office. Moreover, in this script, information is contained on the modes (e.g., calls to external tools) that can be used by the AD to perform some phases of the "official-document" preparation (e.g., entering sample document texts /Fugini 90a/). The meta-level classes provide guidelines for reusing application components of the PAdm framework.

As an example, consider the "document" application class illustrated in Sect. 2: the AD is guided in the selection of this class and of roles thereof through some meta-level roles associated to "document". In particular, associated to each meta- level role is:

- *descriptive information*, concerning relationships among classes;
- *design suggestions*, which drive the design process by expressing actions that the AD must, should eventually, or might perform for customizing the application class. Some of the meta- level roles of "document" are reported in Figure 5b. The meta- functionality role at the meta-level has the same function as the base role at the application level. It defines a basic set of characteristics and functionalities of the document class. Its properties in the example are descriptive information specifying, besides the current level of description, the application domain, the required class(es) necessary to operate the "document" class, its components, and the roles at the application level.
  The other parts of the meta-functionality aim at specifying how the "document" class can be reused. The messages and rules specify that it is possible for the AD to instantiate a meta-level role "meta-signature", driving the AD in creating a "signature-handler" role at the application level.
  This is an example of *design action* that allows the AD to refine a class by tailoring the application level classes through definitions. Another design action is the select-role message specifying that the base role and the being-prepared roles are the default roles when "document" and its subclasses are selected. The default role includes a number of related functionalities, such as creating, filling in, deleting, and so on, as illustrated at the application level (see Sect. 2).

Using meta-level roles, such as meta-signature, the AD can define additional roles to be added to document specifications. The design actions associated to these meta-level roles define the actions that the AD should perform to tailor the class to particular needs (such as, for example, define a list of users for a given document type). Implementation suggestions can also be attached to the properties of a meta-level class referring to implementation level classes that can be used to develop that portion of the application

in the subsequent phases of the ITHACA development phases.
In the example, a list of *standard suggestions* is given among the properties of the meta-functionality role to implement documents in a PAdm application (editors, 4GL, graphical tools, and so on). It should be noted that the definitions contained in Figure 5b are only examples, and that a more complete approach takes advantage of functionalities of the Selection Tools of the SIB. In fact, a query on the SIB can be associated to the "design suggestions" property, and the evaluation of the query would provide the actual design suggestions. Moreover, the AD, in the subsequent development phases, will consider not only the listed suggestions, but also their specializations and possibly similar classes. Meta-level roles for other components of our example are reported in the Appendix.

3.2 Use of frameworks in the specification process

The framework concept illustrated in the previous section is used in the RECAST approach to suggest the AD the design actions that should or could be performed on reusable components to develop an application. Guidance to the AD is provided in terms of messages defined in meta-level roles of the framework meta-level classes; these messages can be invoked by the AD. Each message is presented to the AD in a To-Do list; to each message, some design actions are associated. Actions are translated into queries on the SIB and have two effects:

- enter some definitions in the **requirements document** which is the result of the specification phase and is incrementally filled in with selected reusable classes;
- enter some design suggestions in the **design workspace.**

The retrieval mechanism envisioned for design suggestions from the SIB, where they are stored, is based on querying and navigation. *Interface issues* regarding the SIB will be illustrated in Sect. 4.
Since on of the goals of RECAST is to select from the SIB the design components necessary to script an application, to each selected component a set of justifications is attached. These inform the AD of the reason why a component has been selected in association to a given part of the specification. Design suggestions are provided in the following format:

```
Requirement Component Name + set of keywords +
Script Name + set of keywords +
Application Level Design Actions + Comments
```

This information allows the AD to retrieve relevant scripts and ultimately software components in association with the specifications prepared with RECAST. The association between specification components and scripts is mainly predefined, and stored in the knowledge used by RECAST to guide the AD. Design actions to be performed at the application level (e.g., related to the document class, the use of editors, 4GL, and so

on) are also indicated. We consider three main categories of design actions:

1. component refinements
2. component modifications
3. component interconnections.

1) *Refinements* are used to complete the specifications according to the guidelines provided by the meta-level descriptions.
Refinements can be specified at the meta-level or at the application level. The first ones include the following design actions:

- selection
- instantiation
- definition.

Selection can be: *property selection* (deciding whether a class includes a given property at the application level), *role selection* (deciding whether a class includes a given role at the application level).
Also a framework can be selected by the AD, thus bringing him to a new requirement definition environment, as illustrated later here. Role instantiation occurs at the meta-level: related to the current meta-role, a meta-role can be instantiated for refinement of some design actions.
An example is given in Figure 5b by the design action "instantiate-role meta-signature". Definition (see in Figure 5b create-role "signature-handler") is the action of defining default values for properties and roles. Refinements actions specified at the application level are mainly calls to external design tools. An example is given for the "document" class in its document-preparer role, where document definition tools are specified.

2) *Modifications* are the actions of defining new requirement components, either from scratch or starting from the available ones performing substantial changes to them.
Modifications can be performed along several dimensions:

- *functional requirements*: for each role, the following actions can be performed (not necessarily in sequence):
  - add details (properties);
  - define components: using composition and decomposition mechanisms;
  - specialize a class: specialization may include modifications to the sequences of tasks of a process class, i.e., the modification of the global role state diagram described in Sect. 2.
  - generalize a class: this is a process of assimilation of classes allowing the AE to keep the SIB small and consistent.

- *non-functional requirements*: these include modifications to security, performance, interface, presentation, and so on, that is, to requirements which are not predefined for the reusable classes extracted from the SIB.

- *design tracking*: this type of modifications occurs when design choices made in the subsequent phases of the ITHACA development method (e.g., in the implementation phase) have to be traced back to requirement specifications. It is assumed that it is the responsibility of the AD (possibly via a support tool) to keep track of these cases and perform the necessary actions on the specifications.

3) *Component interconnections* occur according to the model illustrated in Sect. 2 and are specified in the meta- level classes in the rules of meta-level roles. Their general format is:

if DESIGN ACTION then {N/E/P IMPLICATIONS}

where a design action denotes selection, modification or definition of one component, or setting the connection between components. Implications are design actions themselves that need (N), should eventually (E), or can possibly (P) be undertaken.
They include also the connections between components that should be set. In Figure 5b, the implication arising from the SELECTION design action of the "document-preparer" role in the person/office ORM class is that the SELECT-ROLE design action needs to be performed bringing to the "being-prepared" and "being-delivered" roles selection in the document component. Another example is given in the same figure by the rule of the R2 meta-presentation role: if DEFINE official-document then P SELECT-FRAMEWORK official- document.
This is an example of *suggested selection* of a scripted framework, that brings the AD to the environment where he can define, call the suitable tools, and specialize the "document" component. The process of requirement specification is performed by the AD proceeding by different abstraction levels. The idea of guided tours has been proposed for RECAST in /Pernici 90b/. The steps of a *guided tour* drive the AD from a first definition of the application to the detailed specification through a series of design operations which lead to the definition of various drafts of the application. The first draft of the application contains primarily top-level specification components; it specifies the interactions of the application with the external world. By revising and detailing this first draft, various revised schemas of the application are produced. The components are detailed in their structure and behavior; in one schema, classes decomposed at different levels of detail may co-exist. Detailing occurs via refinements and modifications. Refinements are those design operations which are pre-defined at the meta-level; moreover, design and implementation decisions in the subsequent phases of the development are not affected by refinements. Modifications to intermediate schemas are those operations which substantially modify the behavior of the application; modifications can be handled in several ways. If a new type of document, e.g., financial report, is to be handled in the PAdm office, for which no design actions are defined in the document meta-level class, some new components, and subsequently some new software, should be defined.

4 RECAST

In this section we analyze how the concepts presented in the previous sections, and in particular modeling with ORM concepts following the guidelines expressed in a set of frameworks, can be supported by semi-manual and computer based assistants. In Sect. 4.1, we present the architecture of the tools for the assistance of AD, while in Sect. 4.2 we discuss interface aspects.

4.1 Architecture of RECAST

The goal of the tool is to suggest which are the basic design elements to start from, and how to add new features to the application at hand, by taking into account existing possibilities. As a consequence, we expect that the AD will not go through analysis, design and development phases in sequence, but will construct the application incrementally, using development support tools, switching back and forth through the different development phases. In this incremental process, we can distinguish two basic phases:

- the **requirements collection and specification** phase, with two goals:
  1. the definition of the requirements of the application, in terms of real-world entities, and
  2. the selection of the design components that are useful to build the application, giving indications about their expected use;
- the **design phase** has the goal of designing the application in detail, on the basis of the result of the previous phase. Both the requirements specification phase and the design phase are based on information extracted from the SIB, as discussed in the following.

Since the AD develops in parallel different aspects of an application at different levels of detail (given the incremental approach), the system has to maintain the progress of work and the relationships between information represented at different levels. A basic architecture of the ITHACA Application Development Environment is presented in Figure 6. Both the requirements specification and the design tool can work independently. They share common information through a common workspace, which is also the area where the Selection Tools store information retrieved from the SIB. Both private and common workspaces contain temporary information, used during a design session. More permanent information, such as results from the development process, to be stored permanently, is stored directly in the SIB, under the control of the AD.
RECAST works on a particular type of data stored in the SIB, called *Application Description*. Stored information can be retrieved using keyword-based queries or navigation through links, which can be of different types. As described in /Fugini 90a/ and in Sect. 3, classes and frameworks are organized in a semantic network, where the links can either be is-a links or has-part links, as defined in /Koubarakis 89/. This semantic

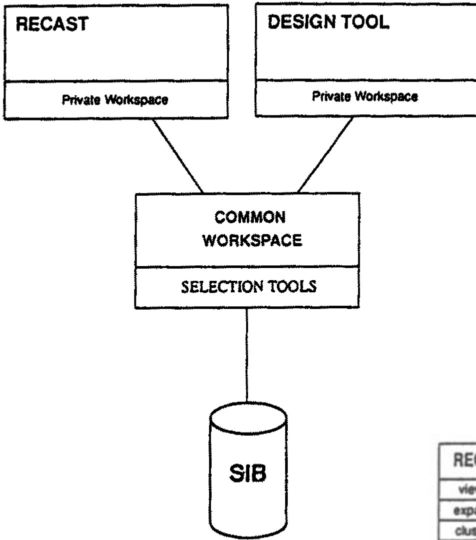
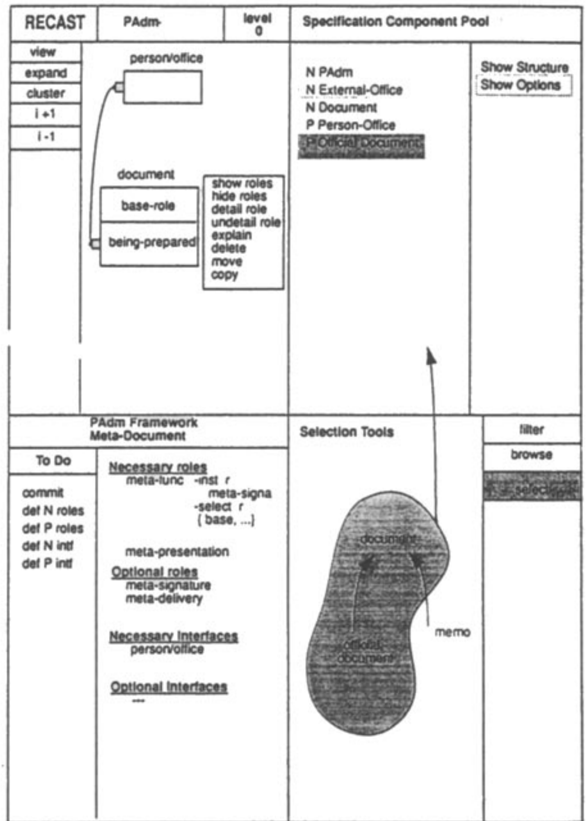


Figure 6. Application Development Environment Architecture



network is, in fact, the basis for the interaction with the AD. Keyword-based queries are used mainly in association with meta- level definitions.
During the specification process, queries are used to access directly a node of the network from which navigation is started. Browsing and keyword-based queries can also be used by the AD, to improve his understanding of the application. For instance, the AD can examine other applications previously developed following correspondence links, find similar classes through similarity links, and so on. The requirements document being produced as a result of the specification process is also stored in the SIB. This document contains design decisions (such as, selection of groups of components, creation of new components, refinements) which have been "committed" by the AD, that is, have been examined, tried in the temporary workspace in terms of components connections, and eventually definitely selected. We distinguish between navigating in the SIB to explore alternatives and selecting components to create the requirements document. Only committed decisions are stored in the SIB, to be used during further steps of the development of the application, or as reference material for future applications. Therefore, we have three types of requirements during the preparation of the requirements document:

- temporary requirements, under exploration locally in a given design phase;
- temporary requirements, shared between development phases;
- requirements stored permanently in the SIB.

From the considerations above, it should be evident that the development of requirements can be supported by *tools* which present very different characteristics. The tools range from completely manual tools to sophisticated and expert design assistance. In fact, it is possible to construct a paper based tool, using a mechanism based on index cards, as proposed in other object- oriented development methods /Coad 90/. Reusable components are stored in a paper file, and retrieved and reused locally, in exploratory incremental design, or in the requirements documents. At the other end of the scale, we envision sophisticated computer-based support, with the assistance of expert tools to check the consistency of, and to automatically complete the requirements document. In the next section, we present an intermediate solution, where some computer-based assistance is provided to the AD based on indications contained in frameworks.

4.2 RECAST interface

The user interface to the development tool in some measure reflects the tool internal architecture. The AD is provided with three windows (each of which can be resized to full screen size), one for each of the basic tools:

- a specification window (RECAST window),
- a design window,
- a selection window.

Within each window, each tool is working independently, and more subwindows are created either automatically or upon the AD's requests. In the ITHACA ADE, the basic structure for the specification and design windows is the same. First, we assume to duplicate the selection window for each of the tools, so each tool can work independently in its selection workspace. Each tool window contains the following information (in this paper, we focus our attention to the RECAST window, illustrated in Figure 7):

- an area for composing components in the *development document*;
- an area for the pool of selected components;
- an area dedicated to provide assistance to the AD.

The problem of *orientation* of the AD within the development information has emerged clearly: existing components must be provided to the AD, however details must be hidden as far as possible, and the appropriate level of abstraction given; irrelevant features should not be presented, the set of elements displayed should be based on the operations been performed, and the detail should be adjustable by the AD at any moment.
To this aim, in RECAST we consider various abstraction levels, as explained in Sect. 3 about specification levels and about drafts refinement steps. The specification window of Figure 7 is divided in four parts. The upper left part allows the AD to interconnect components according to the concepts presented in Sect. 2. Here, person/office has been selected by the AD, and document, with the base-role and the being-prepared role, has been suggested by RECAST (shadowed area). The lower left part presents design guidelines, and the operations allowed to follow these guidelines. The to-do list for the AD allows him to select a define optional (P) roles, necessary (N) roles and to define the non-functional requirements; eventually, the COMMIT option allows the AD to terminate the specification of document. The right quadrants of Figure 7 are devoted to the inspection and selection of the reusable components stored in the SIB. It is possible to select a set of components with the Selection Tools, and to insert them in the pool of specification components. To each component, a meta-level class is associated which provides design guidelines to the AD. The example in Figure 7 is given for "document", the window of the Selection Tools shows the is-a hierarchy of document; here, the AD selects the official-document class, which is therefore moved into the component pool area.

5 Concluding remarks

In this paper, we have illustrated the approach to requirement specification which is currently being implemented within the framework of the ITHACA project. The composition of reusable requirement components, and the concepts of design framework and design guidelines have been presented; the realization of the RECAST support tool has been discussed, in particular in relation to the user interface and to the interface with the other two tools in the ADE. The goals of RECAST can be summarized as follows:

- the specification of an application is a collection of class definitions
- reuse of components is emphasized, and therefore class definitions should be derived from those contained in the SIB
- the AD must be driven in the selection of class definitions from the SIB; to this purpose, each class definition has an associated meta-level description, which contains information useful to the AD to select and modify application classes.The definition of classes is based on the ORM (Objects with Roles Model) model /Pernici 90a/. ORM is based on concepts of object- orientation and its aim is to provide a specifica-tion language for object-oriented applications. The main characteristics of the ORM model are the following: a) to partition functionalities of objects belonging to the same class according to the role played by the object at a given time; b) to provide a rule- based mechanism to model object evolution in time.
  We suggest to use the same role-based mechanism during development for two distinct purposes: a) to drive the construction of the specifications; b) for the specifications themselves.

Further investigation is needed to define a rule-based language to specify implications between components, based on a minimum set of components, and an *expert support* which can be provided on the basis of these rules and of the information associated to classes in meta-level classes. Another important aspect to be investigated is the issue of multiple ADs working in parallel with RECAST. While the case of several ADs working at the same time on different applications is not critical, since we can assume to have a different set of work spaces for each application being developed, the problem of several ADs working on the same application at the same time should be considered. In particular, integration of partial results and interactions between the specification and the design phase are critical /Fugini 90c/. A third important aspect is the fact that components should be provided in a reusable form, that is mechanisms are needed to support design for reusability. A development support tool should be able to automatically provide information associated to new components which facilitates reuse. In particular, it is important to keep track of the design process, and of decisions and motivations for such decisions.

Finally, the mechanism of class retrieval through keywords needs to be further investigated in connection with the progresses made in the Selection Tools ITHACA working group. It is relevant for the AD to retrieve components based on a significant thesaurus of keywords describing the functionalities and development history of components. Broad-scope queries should be coupled to the navigation and browsing mechanims of the Selection Tools; these queries should allow the AD to retrieve similar components and to have an idea of how well a component fits the requirements or if there exist other components that better match the needed functionalities.

Acknowledgements
The authors are thankful to the ITHACA partners in the "Tools Group" for ideas and common work.

References

/Ader 90/ M. Ader, O. Nierstrasz, S. McMahon, G. Müller, A-K Pröfrock, "The ITHACA Technology: a Landscape for Object- Oriented Application Development", *Proc. ESPRIT'90 Conf.*, Kluwer Academic Publishers, November 1990

/Ceri 90/ S. Ceri, P. Wegner and G. Wiederhold," Towards Megaprogramming", Politecnico di Milano *Internal Report* n. 90-055, November 1990.

/Coad 90/ P. Coad, E. Yourdon, *Object-Oriented Analysis*, Yourdon Press Computing Series, 1990.

/Constantopoulos 89/ P. Constantopoulos, M. Jarke, J. Mylopoulos, B. Pernici, E. Petra, M. Theodoridou and Y. Vassiliou, "The Ithaca Software Information Base: Requirements, Functions and Structuring Concepts", *ITHACA Report* ITHACA.FORTH.89.E2.1, 1989.

/Costantopoulos 90 / P. Costantopoulos, M. Theodoridou, M.G. Fugini, "The ITHACA Selection Tool", *ITHACA Report* FORTH.POLIMI.E3.5.1, January 1990

/Cox 87/ B. Cox, *Object-Oriented Programming*, Addison-Wesley 1987

/De Antonellis 90/ V. De Antonellis, B. Pernici, P. Samarati, "Object-Orientation in the analysis of work organization and agent cooperation", Politecnico di Milano, *Technical Report*, (forthcoming).

/Fugini 90a/ M. G. Fugini, B. Pernici, "RECAST: A Tool for Reusing Requirements", in *Advanced Information Systems Engineering*, B. Steiner, A. Solvberg, L. Bergman (eds.), Springer-Verlag Lecture Notes in Comp. Sc., 1990

/Fugini 90b/ M. G. Fugini, B. Pernici, "Cooperative Development of Resuable Design Units", ACM CASE '90 Workshop, Irvine, CA, Dec. 1990

/Fugini 90c/ M.G. Fugini, S. Faustle, M. Theodoridou, D. Vista, D. Nastos, "Technical Description of the Selection Tool", *ITHACA Technical Report*, Polimi.Forth.90.E3.5.#2, January 1990.

/Gibbs 90/ S. Gibbs, D. Tsichritzis, E. Casais, O. Nierstrasz, X. Pintado, "Class Management for Software Communities" , *Comm. of the ACM*, vol. 33, n. 9, September 1990

/Harel 88/ D. Harel, "On Visual Formalism", *Comm. of the ACM*, May 88, Vol 31 n. 5

/Hood 87/ *Hood Manual*, CRI-CISI Ingenierie-Matra, June 1987

/Junod 89/ B. Junod and G. Kappel, "An overview of the TAO office automation system," ITHACA.CUI.89.E.#4, April 4, 1989.

/Kappel 90/ G.Kappel, "Proposed Reference Example for the TWG in ITHACA", ITHACA.CUI.89.E#7 (Revised Version), Sept.1989

/Koubarakis 89/ M. Koubarakis, J. Mylopoulos, M. Stanley, A. Borgida, "Telos: Features and Formalization", Univ. of Toronto *Technical Report*, KRR-TR-89-4, Feb. 1989

/Nierstrasz 89/ O. Nierstrasz, "The ITHACA Application Development Environment - Rationale and Approach", *ITHACA Report* ITHACA.CUI.89.E.#8, May 1989.

/Nierstrasz 90/ O. Nierstrasz, L. Dami, V. de Mey, M. Stadelmann, D. Tsichritzis, J. Vitek, "Visual Scripting Towards Interactive Construction of Object-Oriented Applications", in /Tsichritzis 90/

/Parmigiani 91/ C. Parmigiani, A. Pifferi, B. Pernici, "ORM classes reusability", Politecnico di Milano, *Technical Report* (forthcoming)

/Pernici 90a/ B. Pernici, "Objects with Roles", *Proc. ACM-IEEE Conf. on Office Info. Systems*, Boston, April 1990

/Pernici 90b/ B. Pernici, "Class Design and Meta-Design", in /Tsichritzis 90/

/Petra 90/ E. Petra, "Hypertext Representation of the SIB Descriptions", *ICS-FORTH Technical Report*, July 1990

/Prieto-Diaz 87/ R. Prieto-Diaz, P. Freeman, "Classifying Software for Reusability", *IEEE Software* vol. 4, n. 1, January 1987

/Stadelmann 89/ M. Stadelmann, G. Kappel, J. Vitek, "ITHACA Visual Scripting Tool: A First Implementation Based on the UNIX Shell Scripting Model", *ITHACA Report*, ITHACA.CUI.89.E4.#5, Centre Universitaire d'Informatique, University of Geneva, December 1989

/Tsichritzis 90/ D. Tsichritzis (Ed.), *Object Management*, Centre Universitaire d'Informatique - University of Geneva, Technical Report, July 1990

Appendix

Definitions of some metaclasses of the PAdm example.

```
metaclass person/office

  /* meta-level roles */

    R0  meta-functionality

        properties /* descriptive information */

             domain: PAdm
             level : meta
             required interface: ---
             components: employee, responsible
             appl-level-roles:  (base, requester-for-approval,
                   request-handler,  reminder-informer,
                   doc-preparer, approver)

        messages

             . . .

        rules

             if select-role doc-preparer
                 then
                     N select-role (being-prepared,
                                      being-delivered)
```

```
metaclass PAdm process-class

  /* meta-level roles  */

   RO  meta-functionality

        properties /* descriptive information */

             domain: PAdm
             level : (meta, top-level)
             required   interface:  (person-office, document)
             components: department, dossiers = set-of document

             appl-level-roles:  (base, internal, external,
                                       communication)
             implementation suggestions:
                 (spreadsheet, 4GL, DBMS, scanner, e-mail)


        states  /* global role states */

             (start, suspend, resume, stop)


        rules

             if select-role SAME.external
             then N select-role person/office.external
```

APPLICATION LEVEL

```
  /* top-level specification */

   class PAdm

    base role

        properties /* descriptive information */

             level : top-level
             external interface: client
             components: department, dossiers
             keywords: Public_Administration, Public_Office,
                       Authorization/Request_Handler


        states  /* global role states */
             (start, suspend, resume, stop)
```