Application Development through Reuse: the Ithaca Tools Environment [*]

M.G. Fugini, O. Nierstrasz, B. Pernici[†]

Abstract

This paper presents the architecture and basic features of the Ithaca Application Development Environment based on a Software Information System for enhancing reusability of both software components and artifacts about development of these components. Object-oriented techniques are used in the Environment at all levels of the development of an application: requirement specification, scripting, implementation through class refinement and tailoring. In the Environment, it is tracked how the various products of the development phases were produced by providing tools for the Application Engineer who is responsible for abstracting application skeletons and development information and storing these as Application Frames into a Software Information Base. In particular, the paper describes the Requirement Collection And Specification Tool (RECAST) and the Visual Scripting Tool (Vista) of the Ithaca Development Environment.

*.In ACM SIGOIS Bulletin., Vol. 13, No. 2, Aug. 1992, pp. 38-47.

†.Authors' addresses: Dr. Fugini, Politecnico di Milano, piazza Leonardo da Vinci 32, I-20133 Milano MI, Italy; e-mail: fugini@ipmel1.polimi.it. Dr. Nierstrasz, Centre Universitaire d'Informatique — Université de Genéve, rue du Lac 12, CH-1207 Genéve, Switzerland; e-mail: oscar@cui.unige.ch. Prof. Pernici, Università di Udine, via Zanon 6, I-33100 Udine UD, Italy; e-mail: pernici@uduniv.cineca.it

1 Introduction

The use of Software Information Systems [15] is considered an effective way of achieving some basic goals in software development today. These goals are centered around the idea of reusing software components developed in various projects in order to streamline the development of "similar" applications in selected application domains. The approach based on component reuse needs support both for the process of developing specific applications and for the activity of developing generic, reusable software. In particular, concepts, methods and tools are needed (1) to organize and manage software and information about its development; (2) to identify and reuse useful software components developed in various projects; (3) to manage long-term evolution of reusable software components.

This paper describes the approach to application development considered in the Ithaca (Intelligent Tools for Highly Advanced Commercial Applications) Esprit II Project, started in January 1989 and extending over 5 years [1][‡]. In Ithaca it is considered that the traditional software engineering methods that have worked reasonably well in the past for stable, long-lived applications fail to yield good results today because of two central assumptions that no longer hold [21]: (1) that the application requirements be stable and well-defined, and (2) that the application run in a closed and finite universe. Increasingly, applications are open systems, which can be "open" in each of the three following ways [30]:

1. *Platform:* the hardware and software platform continuously expands.
2. *Interoperability:* open applications must be prepared to exchange information and otherwise interact with systems that may not yet exist.
3. *Evolving requirements:* the application must be flexible enough to adapt to continuously changing requirements.

‡.Esprit II is the second phase of the European Strategic Programme for Research in Information Technologies of the European Communities.

Moreover, a basic assumption of Ithaca is that applications are not unique, but rather share many common functionalities which can be factored out into reusable elements [22]. On this assumption, application generators and fourth generation languages have based their success, however limited. In Ithaca, a global view of the issues related to software development is considered using a Software Information System approach: the Ithaca Application Development Environment (ADE) is being developed as an integrated environment where a software base contains reusable components and development information, and where a set of development tools supports the application developer in finding and combining components to produce a specific application. Reusability considerations of the approach suggest that object-oriented techniques, which are well-suited to factoring out common functionalities into often highly reusable object classes, offer a more general way of allowing many applications in similar domains to share a large part of their code [2].

Reuse of object classes is likely to entail overhead, and application development based on reusable components may turn out not to be a straightforward process of moving along sequences of development steps, but rather may require cycling along various phases, and in particular may require effort in identifying and adapting components. Unfortunately, it is not possible to just sit down and write libraries of instantly reusable classes [22]: the design of reusable classes is necessarily an iterative, evolutionary process [18].

For these reasons, Ithaca proposes a model of software development in which *application engineers* are responsible for developing generic, reusable software for specific application domains, and *application developers* are their clients, reusing not just object classes but also pre-designed *Application*

Frames that guide the developer towards standard ways of constructing applications.

Initially, class libraries and Application Frames may be developed by re-engineering existing applications in an object-oriented way (that is, so as to factor out common functionalities). As application developers encounter more demanding requirements, the software base must evolve to improve its generality and reusability. Application engineers and application developers thus cooperate in a producer/consumer relationship [8].

To support such an *object-oriented software life-cycle*, tools for storing and managing software information are essential in addition to tools to support the development of classes and applications. Ithaca is a 5-year, 100 person-year/year Esprit II Project to build an environment supporting the development of object-oriented applications [1]. The Ithaca environment includes an object-oriented language (CooL [17]) closely integrated with an object-oriented database (NooDLE [17]), in addition to tools to support application engineers and applications developers. A central component in the environment is the *Software Information Base (SIB)* [6][7][26] which stores and manages structured "descriptions" of software (i.e., interface descriptions of object classes, Application Frames, documentation, etc.). The other tools interact primarily by exchanging and manipulating information stored in the SIB. They include [1] a Selection Tool for browsing and querying the SIB; MaX, a monitoring debugger for CooL classes; RECAST, a requirements collection and specification tool; Vista, a visual scripting tool.

In Ithaca, it is considered that application engineers have stocked the SIB with a library of classes and Application Frames clustered by *application domains*. A sample domain is, for example, the "Public Administration" domain [19] that includes classes belonging to Information Systems for office applications, such as the Office, Document, Form, Letter, Employee classes. Public Administration is one of the "demonstrator" domains for Ithaca [21] which has been modeled in a set of classes being used as a common work platform by the various tools.

The application developer is expected to proceed in the following way to produce a specific application:

1. *Select an Application Frame:* using only a rough sketch of the application requirements, the developer searches and browses the SIB to find an Application Frame corresponding to the application domain and requirements of the application being designed.
2. *Select useful classes:* the Application Frame drives requirement collection and specification according to pre-existing, generic specifications and designs, thus guiding the developer in the selection of reusable classes.
3. *Tailor classes:* the selected classes are incrementally modified using design suggestions given by the specification classes; tailoring occurs by supplying parameters or by modifying class behavior through inheritance.
4. *Script application:* link the selected design classes together by means of a "script" that specifies how the objects will cooperate to implement the required application.
5. *Monitor behavior and continuously develop:* test and validate; as requirements change, adapt the application.

As the understanding of the application domain evolves, the application engineer upgrades the contents of the SIB with new reusable components and development information.

The purpose of this paper is to illustrate the overall architecture of the Ithaca ADE. The organization of the paper is as follows. In §2, the Ithaca approach to application development is described and the ADE is introduced. In §3, the approach to requirements reuse in Ithaca is presented, and RECAST, the *REquirement Collection And Specification Tool* supporting the approach is described. In §4, the Ithaca activity of designing through scripting, and the associated *Visual Scripting Tool*, Vista are illustrated. Finally, in §5, concluding observations and further research in Ithaca are given.

2 Ithaca Scenario

The goal of the Ithaca Application Development Environment is to reduce the long-term costs of application development and maintenance for standard applications in selected application domains [1].

By "standard" applications we mean classes of similar applications that share concepts, domain knowledge, functionality and software components. Standard applications are classified in Ithaca according to application domains: the key assumption is that selected application domains can be adequately characterized in order that an individual application can be constructed largely from standard object-oriented software components belonging to that domain. Therefore, achieving reusability of not just software but also of *development experience* is an essential activity of this approach.

The key *benefit* expected of this approach is that applications developed using the Ithaca environment be flexible and open-ended: it should be possible both to develop applications quickly and flexibly and to *reconfigure* applications to adapt to evolving requirements.

This, in turn, implies the need for a different kind of software life-cycle in which the *long-term development* and evolution of reusable software proceeds in parallel with the *short-term development* of specific applications.

Accordingly, we distinguish between two activities:

- *Application engineering*, which refers to the activity of preparing the reusable components, that is, developing a set of Application Frames to be stored in the SIB. An Application Frame consists of reusable application domain specific components and of the guidelines driving the reuse of those components;
- *Application development*, which refers to the development of specific applications by reusing available components through the Ithaca development tools.

Application engineering is the incremental, long-term activity of preparing and maintaining Application Frames, since reusable software can only be developed on the basis of experience gained from the development of specific applications. The

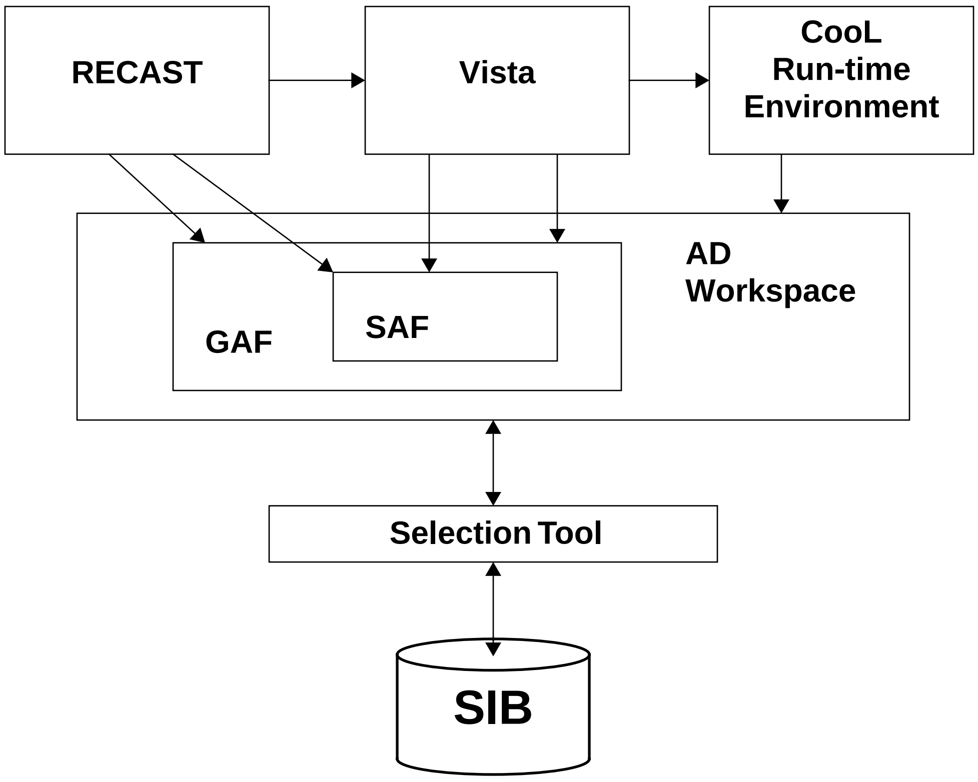

Figure 1 ITHACA Application Development Environment Architecture

benefits of the application engineering activities are expected during application development, since existing requirements schemas, existing designs and existing software can be largely reused.

Most of the activity in application development with the Ithaca tools takes place in the tasks of matching user requirements to generic designs, and in configuring running applications from available components. Since mostly tried and tested software will be used, less time should be spent on detailed debugging, and since construction of applications from existing parts should rapidly lead to evolutionary prototypes, more time can be spent ensuring that the client's needs are properly met. Except for unusual applications, little effort should be spent capturing exceptional requirements, re-engineering existing designs or programming new custom software. Application development is supported by the Ithaca ADE whose overall architecture is depicted in Figure 1.

The **Software Information Base** (SIB) provides the underlying mechanisms for storing, representing, and maintaining reusable component descriptions in the ADE. The SIB elements are structured *descriptions* of software components and their associated requirements and designs in terms of knowledge concerning *application domains*. Since descriptions may refer to one another, the contents of the SIB can be seen as a semantic network, sharing properties of both object-oriented (design) databases and of hypertext systems [24].

The descriptions stored in the SIB are classes representing the results obtained along various software projects. The SIB classes are structured according to four main categories:

- Application Descriptions
- Design Descriptions
- Implementation Descriptions
- Correspondence Descriptions, describing how the three descriptions are inter-related within the development life-cycle, e.g., which requirements relate to which designs and to which implemented objects.

In the SIB prototype, built using the Telos knowledge representation language [20], descriptions are represented internally as Telos propositions, but externally may appear as, for example, software templates, requirements collection forms, or application designs. The uniform internal representation permits advanced queries to be posed to the SIB and evaluated by the Telos inferencing mechanisms.

The **Selection Tool** associated to the SIB provides the means to retrieve software descriptions from the SIB and to navigate through Application Frames [7][13]. The inspection of available development information in the SIB occurs through selection of SIB classes by means of *browsing* and *querying* functionalities.

Browsing is provided to support inspection of the SIB network structure and to explore large collections of data. The browsing facility allows the user to filter link types, to keep track of exploration paths, and to select nodes as "current" to be explored in detail, zoomed in, put in folders, or resumed after a navigation session.

The querying facility provides a formal query language and an application domain based thesaurus. It uses an attribute of SIB descriptions called Functional Description organized according to a software classification schema in the style of [27]. FDs are a combination of keywords describing the functionalities of components and constitute a metric for evaluating the *similarity* of components within the context of application development activity out of reusable elements. The application developer enters a query in the form of a set of weighted keywords describing the functionalities of a searched component; a set of candidate classes is returned, each with a *confidence value* expressing how well the class matches the user search parameters. A *thesaurus* facility is available on-line structured by SIB application domains. A mechanism for maintaining the Functional Descriptions by the application engineer has been defined based on Is-A hierarchies (descriptions are inherited) and based on a Quality Coefficient, which decreases automatically when a returned candidate is discarded by the user.

The Selection Tool returns components into a *common workspace* (see Figure 1), from which each tool imports the components according to its needs and internal format. Conversely, the tools store components into the SIB by placing them in the common workspace; the Selection Tool takes care of the necessary format conversions. The storing of components in the SIB is under the control of the Application Engineer.

In the Ithaca ADE, matching the requirements to existing application domain knowledge stored in the SIB starts immediately as requirements are collected. The **Requirements Collection and Specification Tool (RECAST)** provides “guided tours” of the SIB, attempting to construct a *Specific Application Frame* on the basis of: 1) the user requirements; 2) generic information contained in the Application Frame pertaining to the selected application domain.

Software components selected with RECAST have an associated set of design guidelines describing how components can be reused and what design classes should be selected and scripted in the subsequent phases of the development. In fact, software components selected and identified at this stage are then tailored and composed to construct running applications by a combination of component refinement and scripting activities. The **Visual Scripting Tool (Vista)** [22] supports the interactive construction of applications by graphically editing and connecting visual representations of application and user interface objects [12].

RECAST and Vista are tightly integrated to permit, for example, simultaneously development of parts of the application during requirements collection, re-examination and refinement of requirements specifications during development, and access to programming tools during scripting.

For example, in the Public Administration domain, the Document class is a basic reusable component; RECAST suggests its reuse within specific applications, and gives design suggestions about how this class can be tailored to the needs of the document generation, layout, archiving, and distribution procedures that characterize the application at hand. For each personalization of Document given in the design suggestions, RECAST provides access to design and programming tools such as Spreadsheets, Databases, 4th Generation Languages, and Document Editors.

Both *specification classes* treated by RECAST — which encapsulate requirements and design suggestions — and *scripts* treated by Vista — which encapsulate the bindings between objects — may themselves be reused as components within other specifications and scripts. Furthermore, a script, with the help of the other Ithaca tools, can be viewed as a *generic design*: once an application has been built as a script, it can be unpacked and used for a new application, thus obtaining suggestions for design alternatives via RECAST.

3 Reusing Requirements: RECAST

One of the central research issues in RECAST is the study of a requirement specification method that takes into account the two basic assumptions of the Ithaca ADE, that is:

- the fact that the target application is object-oriented; this recommends the requirement specification should also be performed using an o-o approach [25][9];
- the fact that reusable design components are available; therefore, the development process does not start from scratch, rather reuses as many existing components as possible. The specification phase of the development is affected by the existence of a reusable base of components, and in particular, requirement specifications are reusable components too [14][10].

Within the Ithaca ADE, RECAST (REquirements Collection And Specification Tool), being developed at Politecnico di Milano, enables the Ithaca application developer to specify the requirements of an application taking into account the existing components stored in the SIB. RECAST uses a composition-based approach [2] to requirements specification and provides assisted inspection of available components by accessing the SIB Application Descriptions (see §2).

Moreover, RECAST provides the application developer with development *assistance* about what *design actions* should or might be undertaken by the developer in order to adapt the useful selected components to the requirements of the current application. Design steps, schemas, and tools which can or should be employed in the application development are provided by RECAST as *design suggestions*. This is achieved by providing RECAST with design knowledge stored in the SIB in terms of meta-level classes.

In the following, we illustrate the basic aspects related to the development of RECAST: RECAST knowledge organization (§3.1.) and the main characteristics of the Objects with Roles Model used to formalize such knowledge (§3.2.). Finally, the method supported by RECAST and the tool interface towards the user and towards the other Ithaca tools are presented (§3.3).

3.1 RECAST Knowledge

RECAST uses two levels of knowledge stored in the SIB in terms of Application Description classes:

- application-level knowledge, consisting of reusable *specification classes*;
- meta-level knowledge (design knowledge) representing how specification classes at the application level can be reused and adapted to the requirements of the specific application.

Both the application and the meta-level are composed of classes described uniformly with the formalism of the Objects with Roles Model, and stored in the SIB as Application Descriptions.

Classes and meta-level classes pertaining to the same application domain are clustered into an Application Frame. This allows the developer to consider groups of classes for reusability, instead of single components. The Application Frame drives the user in the development activities; RECAST navigates along a Frame to drive the process of components selection from the SIB, transfer into the application developer workspace and reuse through refinements, modifications, and specializations [14]. The Application Frame has an associated set of *keywords* that allows its identification in the SIB.

Meta-level classes describe: *a)* how the components can be reused, that is, can be selected, refined, modified according to tailoring primitives [10] for being adapted to the application; *b)* what design actions must, or could optionally, be performed by the application developer during the various moments of the design process; *c)* the implications of the design actions performed by the application developer.

For example, in the Ithaca workbench domain of Public Administration Offices [19], two basic reusable components are the ‘document’ and ‘form’ classes. These are returned by RECAST as suggested components, when the designer selects the ‘office’ class. The design actions associated to them specify how these components can be specialized to be adapted to the format of documents handled in Public Administration Offices: for example, which fields should be inserted to model the ‘public event request’ document, which models the documents presented by organizations to get authorizations from the office. As a consequence of the user action of selecting the ‘office’ component and upon the tool suggestion of reusing also ‘document’, the design suggestions contained in the meta-level classes specify that software modules or packages for editing, formatting, filing, circulating documents in the office should also be selected by the developer and linked in the application. The implications of one action are, for example, the connections to the ‘employee’ component modeling the office employee handling the document.

Necessary and optional interface components are also returned as suggested components by RECAST in the local workspace of the developer (see Figure 1): these can be: *a)* required interfaces (components that need to be selected for a given class to work correctly); *b)* component classes, that is, the innermost components of a class providing the class external behavior; *c)* acquaintances (classes that are related to the selected class). Therefore, using the design suggestions incorporated in meta-level classes, RECAST helps to complete the specifications with *default* specification classes and drives the application developer in tailoring the reusable components.

Application-level classes and meta-level classes are represented using the ORM formalism described here beneath.

3.2 The Specification Model

As described so far, the Ithaca approach to application development is fully object-oriented. To this purpose, RECAST knowledge is described using an object-oriented specification model: the Object with Roles Model (ORM) [25]. The ORM model is based on the concepts of object-orientation [4], which are used not only in the Ithaca development language CooL, but also in notations for the analysis and design phases [3][5][28]: the class concept and the inheritance construct.

In ORM, particular attention is given to the life-cycle of objects belonging to a given class. The life-cycle description in a class allows the developer to describe the possible evolution of objects belonging to that class through state transitions. States in ORM are abstract, describing in a synthetic way (with an associated abstract state name) possible situations which an object can be in. Behaviors in a class are described considering separately different aspects in the evolution of an object. To this purpose, ORM introduces the concept of *role*: a role type Ri is defined as a set of properties Pi, a set of operations Oi that can be performed in a role, a set of abstract states Si for that role, and a set of rules Rui, describing state transitions and constraints on properties and possible transitions.

An ORM class is defined as follows:

class = (cname, Roles)

where

$\text{Roles} = \{R_0, R_1, ..., R_n\}$

and

$R_i = \{\text{rname}, P_i, S_i, O_i, Ru_i\}$

The base-role R0 specifies general characteristics of the class, different behaviors and their associated characteristics are described in roles R1,..., Rn.

The ORM model is used in RECAST both at the application level and at the meta-level. At the application level, each ORM class specifies the behavior of possible components of an application. For instance, an office application component in the SIB at the application level is the ‘document’ class, with the following roles: base-role, preparing, delivering, approving, which describe different aspects of the use of a ‘document’ object.

At the meta-level, the guidelines of an Application Framework are described. In RECAST, we associate to each application-level class a meta-level class containing information used to drive the developer in the specification of an application using that class. Such information is partitioned, using the role concept, to consider different concerns separately; for instance, a document has logical contents aspects, layout aspects, distribution aspects that can be specified with different requirements in different specific applications.

Within each role, indications are given to the developer concerning other components to be interfaced with the component being designed, according to different design choices guided by state transitions from a meta-level abstract state to another.

3.3 RECAST composition method and interface

The ability to combine reusable components together in different ways to form different applications is recognized to be a key issue to the achievement of software reusability [2]. Moreover, hierarchical decomposition methods are in widespread use [5], and Hierarchical Object Oriented Design methodologies are proving their effectiveness [16]. RECAST uses a component composition method based on *hierarchical composition* where the behavior of innermost components is encapsulated in higher level abstractions.

Components are connected via *role links* which map outgoing messages of one role into incoming messages of another role and vice versa. Another basic concept for composition is the *Process 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 [9].

Connecting a class role with one selected component via a role link initiates the retrieval from the SIB of the classes and roles whose presence is *Necessary* in the application.

The design suggestions originated by one connection (either explicit or implicitly suggested by RECAST) are the classes and roles Necessary and Optional interfaces. Repeating the connection process, specification classes may be 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. Specification classes may themselves be composed of classes at a deeper level of abstraction; correspondingly, the inner workings of one class is shown in terms of classes providing the class external behavior.

In the example of the Public Administration domain [19], the interaction and inner workings of a sample Public Administration Office with private organizations or persons are modeled, representing institutions (Police Department, Cleaning & Services Department) which might be asked for their authorization when handling public events. Requests and authorizations circulate in the form of documents: request and approval (or rejection) documents. The basic *process class* defined in the SIB for this example is the *'Public Administration'* class; other classes are the 'Person/office', representing the office employees or agents, the 'Document', the 'Official Document', and the 'External Office'. For these specification classes, the required interfaces (Needed classes), the hierarchical innermost components at various *abstraction levels*, and the acquaintances (Optional classes) are defined in RECAST meta-level classes. Specification and meta-level classes appear in RECAST *interface* shown in Figure 2.

In designing RECAST user interface, the problem of orientation in the development information space quickly arises, as usually occurs when dealing with large class collections [15]. Moreover, the architecture of RECAST has to take into account carefully the interaction with the SIB, with the Selection Tool, and with Vista. Therefore, as shown in Figure 2, RECAST provides the application developer with three windows, one for each tool of the Application Development Environment:

- 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 application developer's requests. Each tool window contains the following information (here, we focus our attention to the RECAST window): *a)* an area for combining components in the *development document* [14], eventually containing the set of selected classes; *b)* an area for the pool of selected components; *c)* an area dedicated to provide assistance to the developer.

Existing components must be provided to the user with hidden details to improve orientation: irrelevant features should not be presented, the set of elements displayed should be based on the operations performed, and the detail should be adjustable by the user at any moment. To this aim, in RECAST, various abstraction levels are considered.

The specification window of Figure 2 is divided in four parts. The upper left part is used to interconnect components. Here, person/office has been selected by the developer, 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 allows the developer to select and define optional (P) roles, necessary (N) roles and to define the non-functional requirements; eventually, the COMMIT option allows the developer to terminate the specification of 'document'.

The right quadrants of Figure 2 are the interface between RECAST and the SIB and Selection Tool. It is possible to select a set of components with the Selection Tool and to insert them in the pool of specification components. To each component, a meta-level class is associated which provides design guidelines. In the example in Figure 2, given for 'document', the window of the Selection Tool shows the Is-A hierarchy of 'document'; here, the user selects the 'official-document' class, which is therefore moved into the component pool area.

4 Visual Scripting: Vista

Software components selected and identified at the specification level by RECAST are tailored and composed into the current specific application through the Visual Scripting Tool, Vista.

Visual scripting is an approach to the interactive construction of applications from reusable software components in which both the internal behaviour of applications as well as their user interfaces can be directly edited and manipulated [23]. Vista is being developed as part of the Ithaca ADE by the Centre Universitaire d'Informatique of the University of Geneva.

Vista is based on C++ and X, using the Motif user interface toolkit. The current version of Vista supports a small set of software components developed for demonstration purposes. It is possible to develop, by scripting, very simple forms-based applications that display information retrieved from a database. Transformation on retrieved information can be performed, and information can be propagated from one component to another. A variety of user interface components (buttons, sliders, etc.) can presently be scripted. Ongoing work is to support the use of scripts as components (to permit the use of applications built using Vista as components in larger systems), and, in collaboration with Ithaca partners, the development of other components suitable for scripting.

An exploratory prototype of a visual scripting tool, called VST, which was based on Unix commands and files as compo-

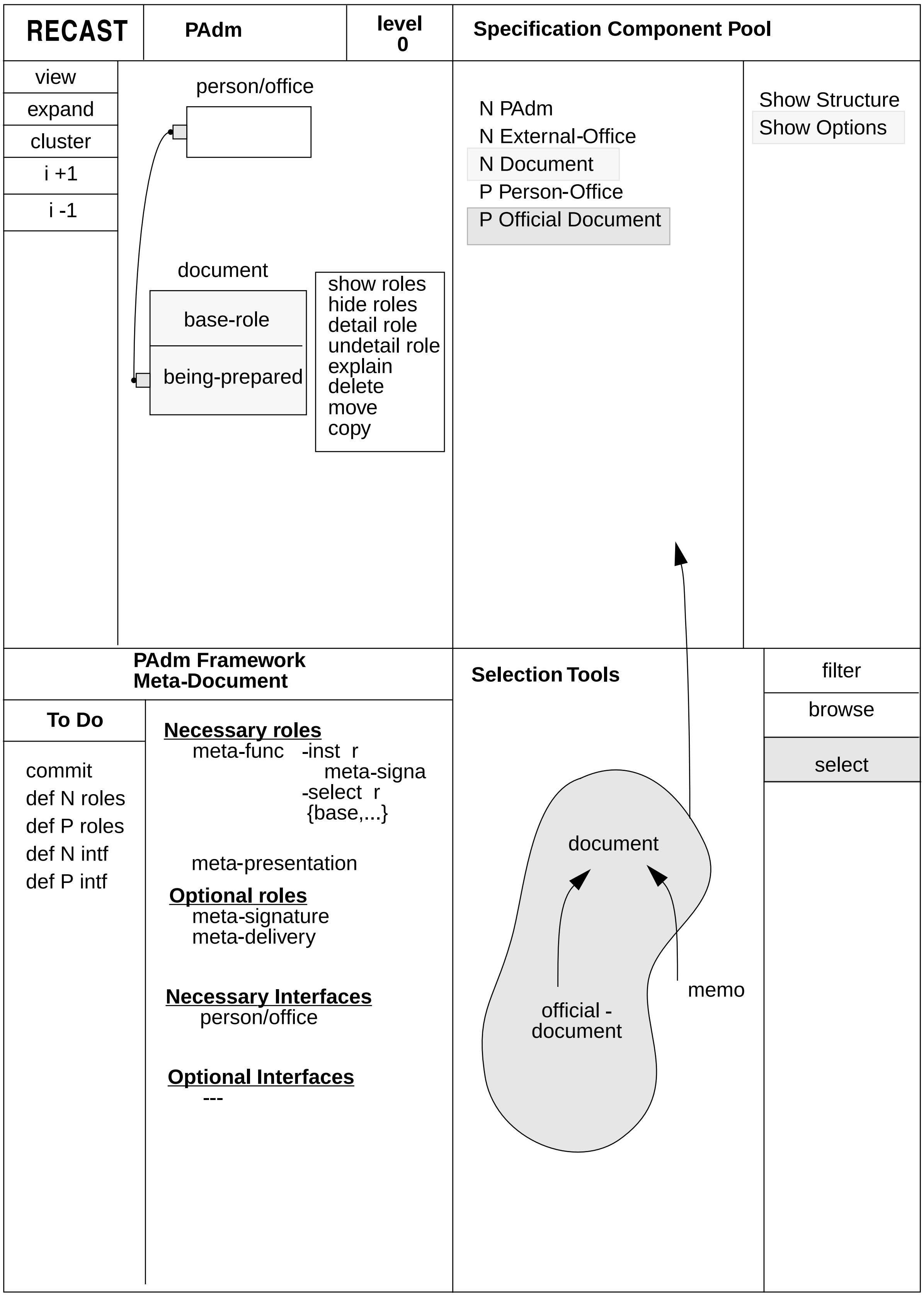

Figure 2 RECAST Interface (from Proc. 3rd Conf. on Advanced Information Systems Engineering, May 1991, with permission of Springer-Verlag)

nents, was implemented during 1989 using Objective C and the ICpak 201 class library [31][29].

VST demonstrated the basic ideas of application development by scripting: the graphical presentation of components; ports to represent the open parameters of components; scripting as a graphical editing activity; encapsulation of scripts as components, permitting their reuse in later scripts; and immediate execution of composed scripts.

Vista is a completely redesigned visual scripting tool that addresses those aspects not covered by the initial prototype, namely: the accommodation of object-oriented software components (rather than Unix commands); more general model of software composition (i. e., other than stream-based connections); support for scripting of user-interface components. In addition, since Vista can be seen as a kind of “graphical editor” for object-oriented applications, it was decided to use the Labyrinth Graph management package [17] developed by FORTH to manage the connections between graphically represented software components.

The current version of Vista implementation [11][12] supports the selection of components by means of a menu, the graphical linking of components, immediate execution of scripts and the saving of scripts. The possibility of using scripts as components is presently being implemented.

A small set of very simple components for demonstration purposes has been implemented: text fields, labels, toggle and arrow buttons, slider, system date, “orchestra”, simple arithmetic calculator, simple database of records, document. The scripting model presently supported is essentially a dataflow model: components make values available on their output ports, and these values propagate to the input ports of connected components. (A “scripting model” standardizes the kinds of ports and links available for a particular components set). This very simple model is sufficiently general to support more advanced scripting models in which connections represent client/server relationships between cooperating objects. The values to be propagated in such cases are the object identifiers — the job of Vista is to keep track of which connections are permissible, and to actually establish the connections at the level of the underlying objects.

A scripting model for *active forms* is currently under development. The model will initially support form and field components, and will later be extended to accommodate other components that store and manipulate forms (e.g., folders, in and out trays, form queries, form producers, etc.). Forms are surfaces that contain field components. Fields are typically “active” in the sense that they may not just store information, but they also have an associated behaviour (verifying a constraint, computing a value, etc.). Furthermore, active fields may control the appearance of a form, which may be associated with several “views” that may change according to the state of the form or who is currently looking at it. Active forms in a sense generalize functionality found in electronic forms systems, hypertext systems and electronic spreadsheets. See [22][23] for an overview.

Work on Vista is continuing now on three fronts:

1. Enhancement of Vista functionality (scripts as components, saving of scripts, improvements to user interface, etc.)
2. Integration with other Ithaca tools (porting to common hardware platform, communication with software information base, scripting of components written in the CooL programming language developed within Ithaca, etc.)
3. Development of demonstration scripting components in collaboration with Ithaca partners responsible for application workbenches (starting with support for scripting of “active forms” — electronic forms whose fields have an associated behaviour, as in, e.g., spreadsheets).

5 Concluding Remarks

The Ithaca Application Development Environment promotes a *component-oriented* approach to software reuse which distinguishes between the activities of (1) preparing reusable software and (2) composing applications from reusable components. This approach is characterized by:

- **An Evolutionary Software Life Cycle:** *application engineering* is the process of producing generic requirements models, designs and software components on the basis of experience gathered from building previous applications; *application development* is the activity of instantiating new applications from the generic designs.
- **Application Frames:** generic and specific application frames structure and organize *software descriptions* at the levels of requirements, designs and implementation.
- **Application Development Tools:** application frames are stored and managed by a *Software Information Base* (SIB) and an associated *Selection Tool* (ST); a *Requirements Collection and Specification Tool* (RECAST) negotiates the “guided tour” of the SIB during the application development activity; and the *Visual Scripting Tool* (Vista) provides a direct manipulation graphical editor for interactively constructing running applications from retrieved and instantiated components.

The key assumption of the Ithaca approach is that a heavy investment in application engineering is justified by the improved configurability, robustness and openness of the resulting applications. The quality of application engineering can be evaluated in terms of how well reuse is supported during subsequent application development. For this reason the SIB is more than just a library of object classes. To support the negotiation activity of RECAST, knowledge concerning the potential for reuse at all levels is encoded in Application Frames in terms of the *Objects with Roles Model* (ORM). This knowledge is used to guide the application developer using RECAST along a certain development path. To support the software composition activity of Vista, object classes are designed to conform to a *scripting model* which standardizes their interfaces and improves their potential for reuse.

The Ithaca ADE will be evaluated for selected applications in three “demonstrator” domains: Public Administration, Financial Applications, and Office Systems. The main difficulty is that it is necessary to play the role of Application Engineer to populate the SIB before the approach and the tools can be tested. It is for this reason that relatively well-understood application domains have been chosen for the initial experiments. At the same time, the approach will have a demonstrable advantage if it can cope well with change and evolution, so it has been important to select application domains in which configurability and openness are implicit requirements. At this point, prototype implementations of all of the tools have been completed and integration work is ongoing.

A longer-term goal of Ithaca is to turn the scenario of the evolutionary software life cycle into a true methodology that incorporates both application engineering and application development. Such a methodology would be based primarily on redesigning and reengineering of existing applications, and would take into account reorganization of Application Frames to incrementally improve reuse potential.

Acknowledgments

The authors are thankful to all the participants in the teams working in the Ithaca Tools Working Group for common work and ideas: Université de Genève, FORTH (Crete), Politecnico di Milano, Siemens-Nixdorf.

References

[1] M. Ader, O.M. Nierstrasz, S. McMahon, G. Müller and A-K. Pröfrock, "The ITHACA Technology: A Landscape for Object-Oriented Application Development," Proceedings, Esprit 1990 Conference, pp. 31-51, Kluwer Academic Publishers, Dordrecht, NL, 1990.

[2] T.J. Biggerstaff and A. J. Perlis, *Software Reusability — Vol. I — Concepts and Models*, Frontier Series, ACM Press, 1989.

[3] G. Booch, *Object-Oriented Design*, Benjamin-Cummings, 1991.

[4] *Special Issue on Object-Oriented Design*, Communications of the ACM, vol. 33, no. 9, Sept 1990.

[5] P. Coad and E. Yourdon, *Object-Oriented Analysis*, Computing Series, Yourdon Press, 1990.

[6] 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.

[7] P. Constantopoulos, M. Doerr, E. Pataki, E. Petra, G. Spanoudakis and Y. Vassiliou, "The Software Information Base — Selection Tool Integrated Prototype," Ithaca Report — ITHACA.FORTH.91.E2.#3, 1991.

[8] B. Cox, *Object-oriented Programming*, Addison-Wesley, 1987.

[9] V. De Antonellis, B. Pernici and P. Samarati, *Object-orientation in the analysis of work organization and agent cooperation*, November 1990 (submitted for publication).

[10] V. De Antonellis, B. Pernici and P. Samarati, *F-ORM-METHOD: a F-ORM methodology for reusing specifications*, January 1991 (submitted for publication).

[11] V. de Mey, "Vista User's Guide," ITHACA Report — ITHACA.CUI.90.E.4.#2, Dec, 1990.

[12] V. de Mey, "Vista Implementation," ITHACA Report — ITHACA.CUI.90.E.4.#1, Dec, 1990.

[13] M.G. Fugini and S. Faustle, "Similarity queries for class selection from a Software Information Base," Ithaca Report — ITHACA.POLIMI.90.E3.6.#1, December 1990.

[14] M.G. Fugini, M. Guggino and B. Pernici, *Proc. 3rd Nordic Conf. on Advanced Information Systems Engineering — CAiSE '91*, Trondheim, May 1991.

[15] S.J. Gibbs, D.C. Tsichritzis, E. Casais, O.M. Nierstrasz and X. Pintado, "Class Management for Software Communities," Communications of the ACM, vol. 33, no. 9, pp. 90-103, Sept 1990.

[16] *HOOD Manual*, CRI-CISI_Ingenierie-Matra, June 1987.

[17] *ITHACA 1989 Milestone, Vol. 2: Work Package X/L — Language and Execution Environment*, Feb. 1990.

[18] R.E. Johnson and B. Foote, "Designing Reusable Classes," Journal of Object-Oriented Programming, vol. 1, no. 2, pp. 22-35, 1988.

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

[20] M. Koubarakis, J. Mylopoulos, M. Stanley and A. Borgida, "Telos: Features and Formalization," Univ. of Toronto Technical Report — KRR-TR-89-4, February 1989.

[21] O. Nierstrasz, "The Ithaca Application Development Environment — Rationale and Approach," Ithaca Report — ITHACA.CUI.89.E.#8, May 1989.

[22] O.M. Nierstrasz, L. Dami, V. de Mey, M. Stadelmann, D.C. Tsichritzis and J. Vitek, "Visual Scripting – Towards Interactive Construction of Object-Oriented Applications," in *Object Management*, ed. D.C. Tsichritzis, pp. 315-331, Centre Universitaire d'Informatique, University of Geneva, July 1990.

[23] O.M. Nierstrasz, D.C. Tsichritzis, V. de Mey and M. Stadelmann, "Objects + Scripts = Applications," draft, Centre Universitaire d'Informatique, University of Geneva, Feb. 1991.

[24] K.G. Pankaj and W. Scacchi, "On designing hypertext systems for information management in software engineering," Hypertext '87 Papers, Nov. 1987.

[25] B. Pernici, *Proc. ACM-IEEE Conf. on Office Info. Systems (COIS)*, Boston, April 1990.

[26] E. Petra and C. Vezerides, "SIB Contents' Manual," Ithaca Report — ITHACA.FORTH.91.E2.#4, ICS-FORTH, January 12 1991, Draft of Version 1.0.

[27] R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," IEEE Software, vol. 4, no. 1, January 1987.

[28] S. Shlaer and S. J. Mellor, *Object-Oriented System Analysis — Modeling the World in Data*, Yourdon Press Computing Series, 1988.

[29] M. Stadelmann, G. Kappel and J. Vitek, "VST: A Scripting Tool Based on the UNIX Shell," in *Object Management*, ed. D.C. Tsichritzis, pp. 333-344, Centre Universitaire d'Informatique, University of Geneva, July 1990.

[30] D. Tsichritzis, "Object-oriented Development for Open Systems," Information Processing '89 (Proc. IFIP '89 Conf.), North-Holland, S. Francisco, August 1989.

[31] J. Vitek, B. Junod, O.M. Nierstrasz, S. Renfer and C. Werner, "Events and Sensors: Enhancing the Reusability of Objects," in *Object Management*, ed. D.C. Tsichritzis, pp. 345-356, Centre Universitaire d'Informatique, University of Geneva, July 1990.