# The Software Information Base: A Server for Reuse

*Panos Constantopoulos *, Matthias Jarke †,*
*John Mylopoulos ‡, Yannis Vassiliou **

**ABSTRACT**

We present an experimental software repository system which provides organization, storage, management, and access facilities for reusable software components. The system, intended as part of an applications development environment, supports the representation of information about requirements, designs and implementations of software, and offers facilities for visual presentation of the software objects. The paper details the features and architecture of the repository system, the technical challenges and the choices made for the system development along, with a usage scenario which illustrates its functionality. The system has been developed and evaluated within the context of the ITHACA project, a technology integration, software engineering project sponsored by the European Communities through the ESPRIT programme. The aim of the project is to engineer an integrated reuse-centered application development and support environment based on object-oriented techniques.

* Institute of Computer Science, Foundation of Research and Technology – Hellas, P.O.Box 1385, Heraklion, Crete, 71110 Greece, e-mail: {panos|yannis}@ics.forth.gr .

† Lehrstuhl Informatik V, RWTH Aachen, Ahornstr. 55, W-51000, Aachen, Germany, e-mail: jarke@informatik.rwth-aachen.de .

‡ Department of Computer Science, 6 King's College Road, University of Toronto, Toronto, Ontario, Canada M5S 1A4, e-mail: jm@cs.toronto.edu .

## CONTENTS

# 1. Introduction

Software reuse has grabbed center-stage in international software engineering research, promising to deliver the productivity increase that will eliminate, or at least alleviate, the software crisis. Unfortunately, the path that leads to reuse is not as clearly defined as its promised results. Software libraries, properly populated, are certainly a step in the right direction. So is organizational support and encouragement of reuse, aided by rewards for experience-sharing among software development teams. Object-oriented computing constitutes yet another touted path to the reuse silver bullet. Better understanding of the process of software reuse, supported by appropriate tools is another. So is research that advocates linking software reuse to design reuse in general (as in hardware or architectural design) and developing general AI-based methods such as case-based reasoning and case-based knowledge organization to address it.

Despite this wealth of diverse approaches to reuse, some themes are common. Fundamental among them is the thesis that reuse concerns more than software code. Designs, requirements specifications, development processes are also reusable and can contribute as much to the legendary productivity increase as the reuse of existing programs. Indeed, software reuse concerns all aspects of the software development experience. Consequently, one can characterize the degree of reuse in terms of a channel of communication between the original developers and the re-users. The broader and better defined the channel, the greater the potential for reuse, and therefore for productivity improvements. For program libraries, for example, the channel is well defined but narrow, since all development experience other than coding is missing. For experience-sharing meetings between original developer and reuser the channel is broad but ill- defined as it relies on human memory. A major concern of research on reuse is the development of methods and tools which broaden and sharpen this channel, by facilitating the recording of the software development experience in all its breadth and richness and by assisting in its selection and adaptation to new software development tasks. A key component for that is repository technology.

Broad and comprehensive surveys of reuse and the technical challenges it poses can be found in [BiRi89, BiPe89, Krue92]. Lines of research address problems such as: designing-with-reuse, designing-for-reuse, software artifact classification (characterization), selection/comprehension of reusable objects, adaptation, etc. Krueger presents in [Krue92] a taxonomy of reuse methods in terms of their ability to abstract, select, specialize (or adapt), and integrate the software artifacts (by composition or combination into a new system.) Even though repository technology is still believed to be very immature, there are some major commercial efforts and platform standards, notably the IBM Repository Manager/MVS, Digital's CDD Cohesion, PCTE+ OMS, CAIS, and IRDS ([Jone92], [Jobe90]). There also exist a host of less ambitious products, like ADW/MVS,

CASE*Dictionary, DB Excel, or Brownstone [Plot92]. In research, there are a number of projects of narrower scope that experiment with applications of traditional or object-oriented database technology ([Ditt87], [HuKi89]), with hypertext environments ([GaSc87], [Bige88], [GaSc89]), and with Artificial Intelligence techniques ([Deva91], [Meye85], [Oste92], [Jark93]). Regarding repositories for source code fragments only, much progress has been made with object-oriented class libraries which provide powerful browsers (Objectworks, SPARCworks Professional C++, CodeCenter, Classix, Eiffel, C++ SoftBench Toolset, etc.)

This paper presents a software repository system designed for reuse as a means towards broadening and supporting the communication channel between developer and reuser. The system is intended to store and manage *information about* requirements, designs and implementations of software and offers facilities for locating and selecting software components. The repository system has been developed within the context of the ITHACA project, a software engineering project sponsored by the European Communities through the ESPRIT programme, whose aim is to develop a complete integrated application development and support environment based on object-oriented techniques. The ITHACA environment includes an object-oriented programming language and database service, as well as application development and application support tools. In all aspects of the project, ITHACA adopts a reuse- oriented methodology. Much of the ITHACA work on languages and tools is therefore focusing on how to make reuse a practical technology. The rest of the paper offers an account of the structure and implementation of the software repository system, or Software Information Base (hereafter SIB) as a means towards reuse.

Assuming a repository-based reuse methodology, key technical challenges - directly related to the repository system development - are those of providing the right abstraction concepts/mechanisms, carefully organizing, effectively managing, efficiently selecting and understanding the software artifacts. Figure 1.1 offers a simplified view of the repository-based reuse process; important functions (e.g., composition of the software artifacts into a new system) are the responsibility of application development tools acting as clients of the repository system.

A number of considerations had to be kept in mind in designing the SIB and its functionality. Firstly, storing information other than code about a software system, such as requirements, designs, design decisions and justifications means that higher-level software specifications may be reused directly, but also that they can serve as indices to lower-level software artifacts, i.e., code. Secondly, the issues of representation and presentation of information about reusable artifacts do not adhere to simplistic solutions and certainly need to be addressed separately. Software artifacts should be treated as multimedia objects, which are created and used by distinct application development tools, and must be finally presented in the form they are expected. A data flow or an entity relationship diagram, for instance, should appear as expected to the tool that uses it. Yet, all these drastically different artifacts need to, somehow be abstracted and represented in a uniform way within the repository to facilitate the user's conception of the contents of the repository and her understanding of the supporting tools. This calls for a common SIB representation, extensible to accommodate new types of artifacts (say, SADT diagrams) along with support for multiple presentation forms, depending on the tool using a particular artifact.

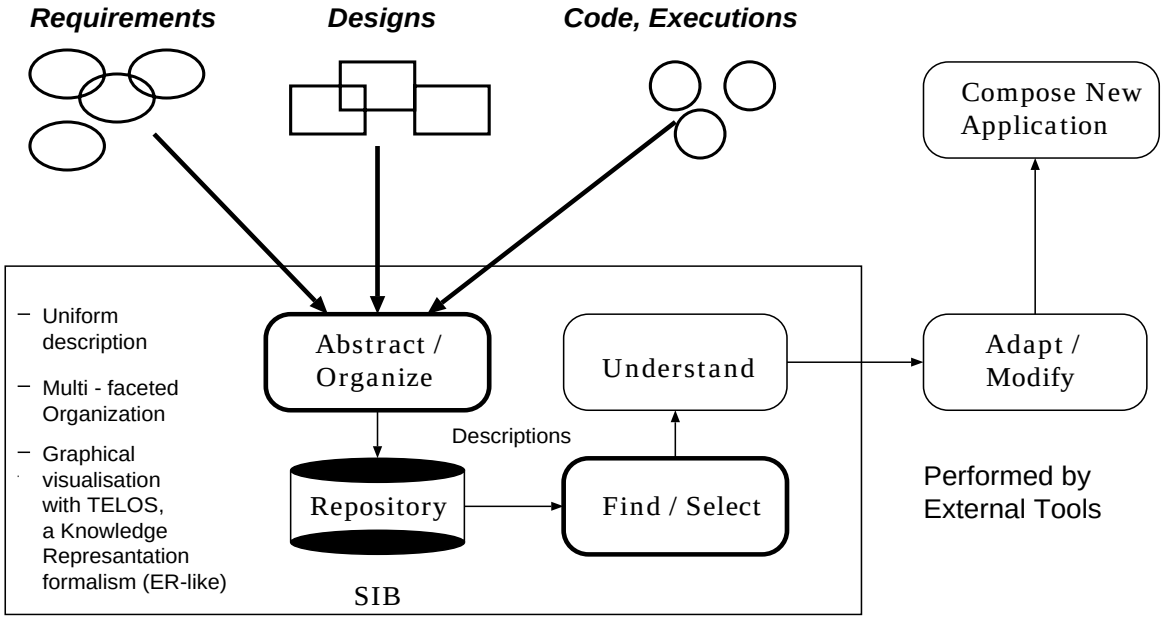


Figure 1.1: The Reuse Process

The representation language chosen for the SIB is Telos [Mylo90]; a conceptual modelling language in the family of entity-relationship models [Chen76], designed specifically for information system development applications. The main reason for the adoption of Telos over other extensions of the E-R model, such as those used by the IBM Repository Manager/MVS or PCTE+ OMS, are its treatment of attributes as first-class objects and the treatment of metaclasses, which together lead to a notation that is both expressive and readily extensible. Moreover, Telos has been shown to have a simple and elegant formal semantics in which the Telos data structures and abstraction mechanisms (including its extensibility) are specified in terms of a deductive relational database using only a few basic system facts, deduction rules, and integrity constraints [Jeus92]. This simplicity offers advantages over existing object-oriented DBMSs especially when designing multiple related query interfaces -- a main feature of the SIB. Specifically, the Telos semantics enables the SIB to offer three integrated query interfaces that might be used by a user at different times, depending on the task the user is working on: a graphical network interface, a form-based interface and a linear (e.g., SQL- or QBE-like) query language like the ones found in relational databases. The disadvantages of Telos with respect to traditional object-oriented DBMSs -- limited integration of procedural methods for complex update operations -- play a lesser role in the SIB context which is very much search-intensive. From a practical viewpoint, the multi-paradigm interface support offered by Telos is made possible through the use of a powerful graphical editor [Kate90] and a hypertext engine. Navigation along links, search through keywords and structured presentations of the SIB contents are all supported as aids for the SIB user. The adoption of both hypertext and conceptual modeling facilities and their integration into a single user interface is intended to alleviate inherent problems of hypertext systems, such as user disorientation and cognitive overhead.

The SIB organization constitutes one of the keys to its usefulness. All querying facilities (browsing, filtering, navigating, etc.) are supported and facilitated by organization principles, which include the usual general purpose organizational principles of conceptual modeling (i.e., classification, generalization and aggregation) and are enriched with principles of modularization, semantic similarity and others. These principles are intended to address both user and methodological needs. For example, similarity links are there because users find it natural to ask for "similar" components, intended to reflect, however, the ITHACA way of constructing software by "scripting" together existing classes [Tsic91], shared by other object-oriented viewpoints where software composition has replaced the problem decomposition methodologies associated with structured programming.

Considering its size, complexity and required investment, effective management of the SIB is obviously critical. The SIB has been implemented as an efficient and stable prototype. Persistent storage, together with much of the functionality found in relational or object-oriented database systems are realized with a, specialized for software repository systems, SIB object management component.

A major goal of reuse is, of course, to "...find the software artifacts faster than the time it takes to develop them..." [Krue92]. Therefore, in addition to providing basic retrieval optimization mechanisms, our approach adopts a multi-paradigm selection strategy, which includes query processing, browsing, filtering, navigational facilities and approximate retrieval based on similarities among software artifacts. Furthermore, the user interface has been carefully designed to meet the challenge of offering in a user-friendly fashion a combination of several retrieval modes. Finally, the SIB system includes built-in facilities which make it possible to associate with any software object annotations and/or animations.

Apart from general considerations concerning the maintenance of any large repository, the context of reuse-based software development introduces a number of additional complications. In particular, there has to be provision for information acquisition, integrity enforcement strategies, version management and schema evolution. We are only beginning to address these and other issues concerning support for SIB users, taking as starting point the prototype implementation reported here.

Finally, while not claiming to address issues concerning software artifact understanding, along the lines of research efforts such as [FiKe92], we do claim that the SIB offers valuable assistance to software artifact understanding efforts through the representation and organization of software descriptions.

The remainder of the paper is organized as follows. Section 2 offers an elaboration on the SIB structure, defined in terms of descriptions which serve as basic building blocks, along with a number of link types used to organize the contents of the SIB. In Section 3, the SIB system is detailed, with emphasis on the rationale behind the choices made for the development. Section 4 presents an empirical evaluation of the approach in the context of a specific reuse-oriented methodology, that of ITHACA, including a sample usage scenario. Conclusions are drawn and future research directions are outlined in Section 5.

# 2. Structure of the Software Information Base: The SIB Model

In a nutshell, the Software Information Base is structured as a directed attributed graph, with nodes describing software artifacts (objects) and edges representing semantic relationships that hold among them. The software objects themselves are assumed to have their own representation, external to the SIB -- say, in terms of a UNIX file storing a C program or an *SADT™* [1] diagram [Ross77] -- which is accessible from the corresponding SIB description.

The basic building block for the SIB is a *description* which provides information about a software system. This information may concern a requirements, design or implementation specification for a particular software system. Descriptions may also be used to represent design decisions or run-time performance information about a software object. In addition, descriptions may be atomic, built up from primitives such as programming language expressions, or composite, having other descriptions as parts.

The modeling constructs (types of links) used in the SIB can be distinguished into four categories:

(I) *General structural/semantic relationships*, including attribution, classification and generalization. These are the basic structuring mechanisms offered by Telos.

(II) *Special structural/semantic relationships*, including aggregation, correspondence, genericity and similarity. These have been identified as a minimal set of system-supplied special software descriptors.

(III) *User-defined and informal links*, including versioning, hypertext, etc. The attribute definition facility supported by Telos can be used to extend this set of links.

(III) *Associations*. Orthogonal to the above binary linking mechanisms, are groupings of software artifact descriptions into larger functional units. Associations are sets of descriptors along with private symbol tables, which allow for the partition of the SIB into coherent subspaces through the creation (in terms of queries) of materialized views (or snapshots) and of contexts (or workspaces).

[1] SADT is a registered trademark of Softec Inc.

## 2.1. Attribution and Aggregation

Descriptions can be related to others through a number of built-in semantic relationships. These include attribution and aggregation (part-whole) as in

```
Description SoftwareObject with
     attributes
          author: Person
          dateOfFirstVersion: Date
          currentVersion: VersionNumber
     hasParts
          components: SoftwareObject
end SoftwareObject
```

Here **SoftwareObject** is defined as a description having three attributes, named `author`, `dateOfFirstVersion` and `currentVersion`, whose 0 or more values have to be instances of the descriptions `Person`, `Date` and `VersionNumber` respectively. **SoftwareObject** also has 0 or more parts of type `SoftwareObject`. Schematically, this description can be represented in terms of the diagram shown in Figure 2.1, with boxes representing SIB nodes and arrows representing SIB edges.

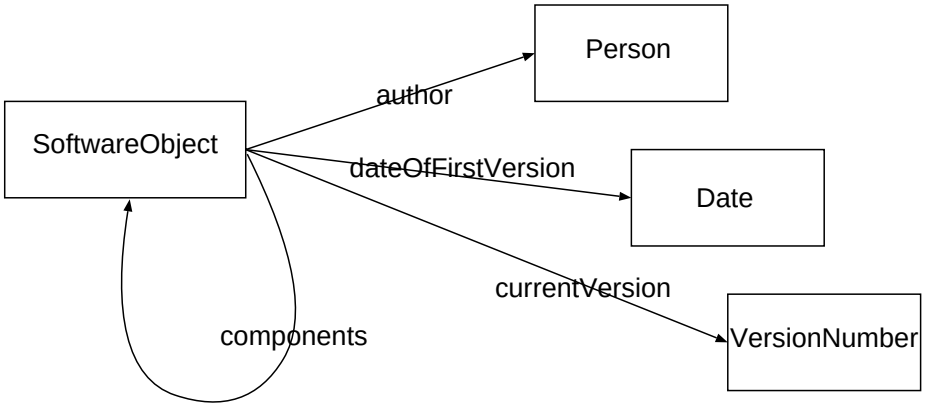


Figure 2.1: Attribution and Aggregation

*Attribution* (based on the notion of attribution in [Mylo90] and similar to the notion of attribution in Omega [Atta81]) provides a general and rather unconstrained representation mechanism which describes an object in terms of attribute-value pairs. *Aggregation*, on the other hand, is a meronymic relationship which relates an object to its components [Wins87]. The components of an object are assumed to be of the same kind (in our case, software object descriptions) and to play an integral role in the object's function. Moreover, changes to a component (for example, through the creation of a new version) imply changes of the aggregate object as well. This property is, in fact, not shared by attributes. It is noted that aggregation axioms imply referential integrity.

## 2.2. Classification

From a modeling perspective, *classification* (opposite *instantiation*) is perhaps the most important semantic relationship represented in the SIB. Every SIB object must be an instance of one or more other, generic SIB objects, referred to as classes. Classes are themselves instances of yet more generic classes (called metaclasses). Most SIB objects are assumed to lie on a unique classification level ranging from 0, for tokens with no instances of their own, to level 1, for simple classes with instances from level 0, to level 2 for metaclasses (or M1 classes) with instances from level 1 etc. Certain SIB objects that take instances from several levels are known as omega classes and belong to the omega-level. Omega classes are needed to avoid infinite regress in the semantics of the language. From a pragmatic point of view, omega classes help define built-in classes such as Proposition (having all SIB objects as instances) and Class (having all classes as instances), hardcoded into the SIB system, and responsible for elements of the system's operational semantics. Figure 2.2 shows the structure of the classification dimension.

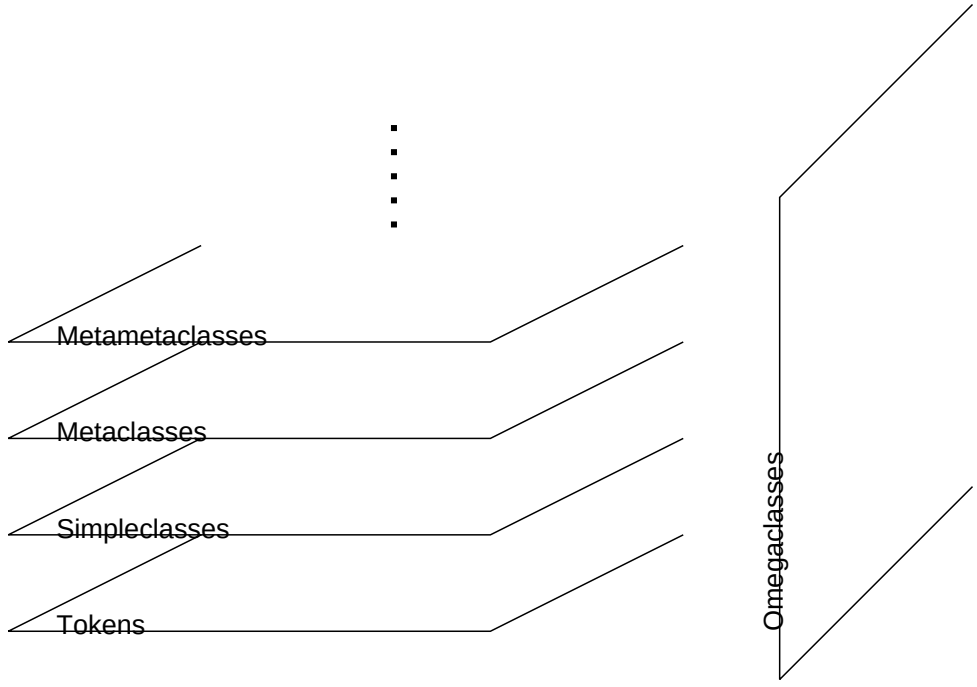


Figure 2.2: Classification Levels

As with other classification mechanisms described in the literature, instantiation of a class involves instantiation of all associated semantic relationships. For example, instantiation of the `SoftwareObject` class involves defining 0 or more instances of its attributes and parts, as shown in Figure 2.3. Note that in the representation adopted, semantic relationships are treated as objects in their own right which are instantiations of relationship classes. Moreover, interrelated objects need not lie on the same classification level.

**HotelIS**, for instance, may be a simple class (with particular execution of this information system as instances), while `6.03.1`, `Sept90` and `Yannis` are tokens.

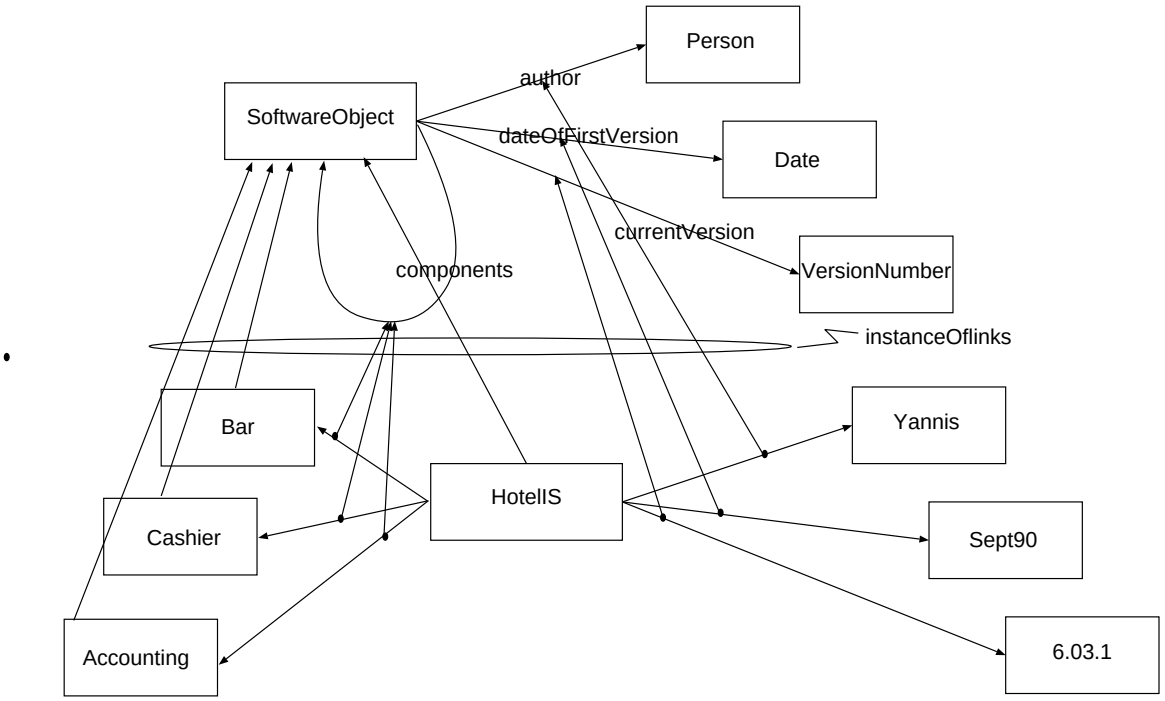


Figure 2.3: Classification Example

Syntactically, the above instantiation is specified in terms of the following:

```
Description HotelIS in SoftwareObject with
     author
          : Yannis
     dateOfFirstVersion
          : Sept90
     currentVersion
          : 6.03.1
     components
          : Accounting, Cashier, Bar
          ...
end HotelIS
```

Note that the edges associated with **HotelIS** may have their own labels, as in

```
Description HotelIS in SoftwareObject with
          ...
     components
          acc: Accounting
          cash: Cashier
          bar: Bar
          ...
end HotelIS
```

In the latter definition of **HotelIS,** its components can be accessed by traversing the edges that are instances of the components link of `SoftwareObject`, or by traversing the edges labeled respectively `acc, cash` and `bar.`

## 2.3. Generalization, Genericity and Correspondence

Those three link types have been grouped together because they have similar definitions which include inheritance. Thus, a class lower down one of these hierarchies has fewer possible instances and its instances inherit from their more general ancestors. Their differences will become apparent as we discuss them in turn.

*Generalization* (opposite *specialization*) has traditionally been supported by semantic and object-oriented data models as well as knowledge representation schemes. The notion of generalization adopted here allows multiple, strict inheritance. For example, the data class `StudentEmployee` can be declared as a specialization of both classes `Student` and `Employee,` thereby inheriting attributes and parts from both classes (multiple inheritance). However, the definition of `StudentEmployee` cannot override any of the inherited information, only further constrain it. For instance, if `age` has been declared to be an attribute of `Student` with type the integer range 3..60, the `age` attribute of `StudentEmployee` might be further constrained to fall in the range 15..45, but cannot be redefined to have a range, say, 15..70. In programming languages, the term *subtyping* has been used for generalization. Note, however, that when an object is a subtype of another, this does not imply that they share implementation.

*Genericity* (opposite *specificity*) links relate software descriptions and is intended to convey the sense that one class is an incremental modification of another. A good example of genericity can be found in programming languages where *implementation inheritance* has been used to represent a situation where a software object is more parameterized, and hence has greater genericity, than another with the implication that at code level the two share (some of) their implementation methods. Like generalization, genericity is assumed to be acyclic, transitive and non-reflexive.

In general, the SIB will include several associated descriptions for a single software object. These may include zero or more versions of a requirements, design and implementation description. A *correspondence* link represents firstly information concerning the identity of the software system described by two descriptions. In addition, correspondence links can have parts which represent structural correspondences among the components of corresponding descriptions.

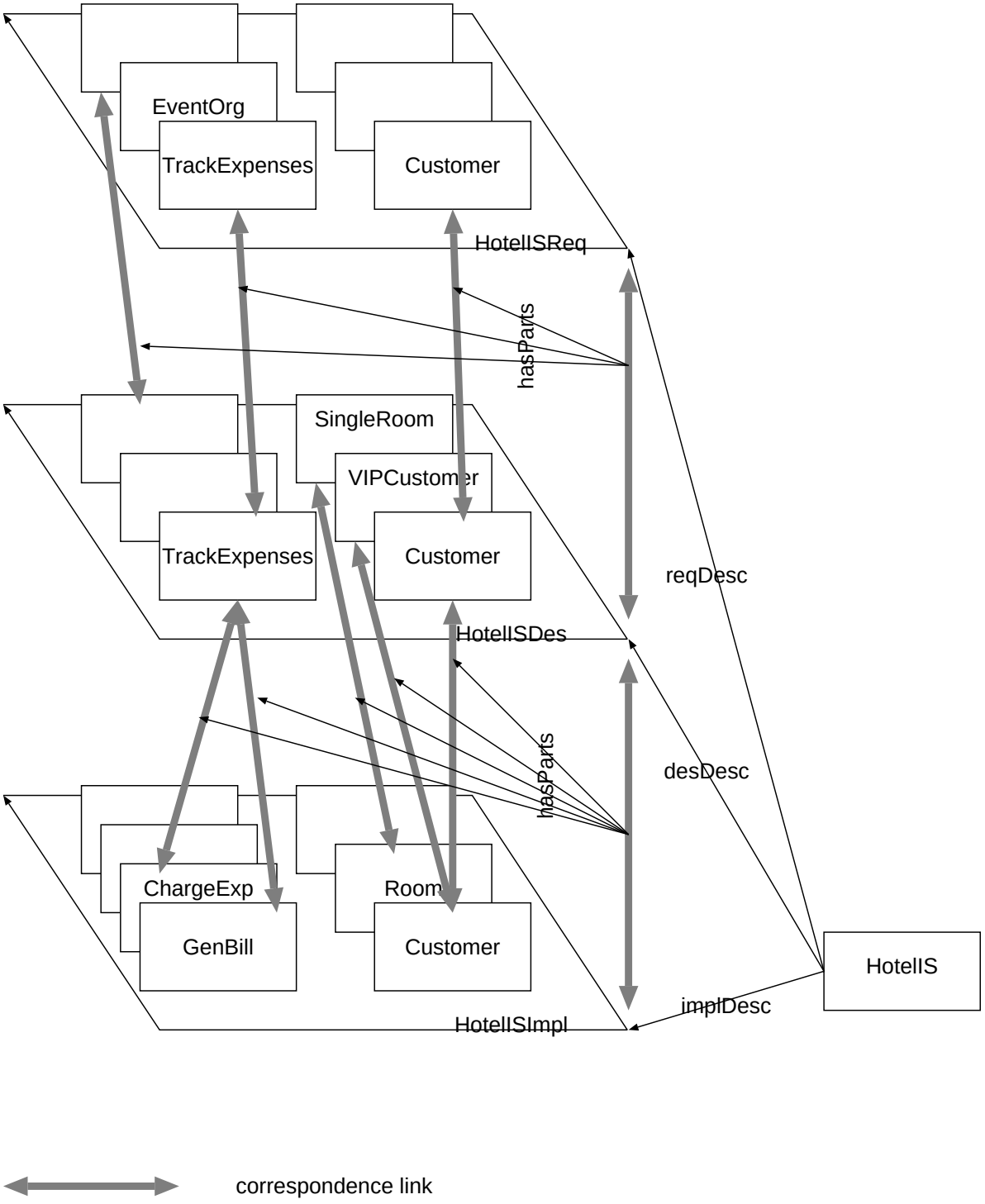


Figure 2.4: Correspondence Relationships

In Figure 2.4, for example, `HotelISReq, HotelISDes, HotelISImpl` represent the requirements, design and implementation descriptions of a hotel information system, described through the HotelIS description. Correspondence links indicate the fact that the three descriptions respresent the same system at different levels of abstraction. These links have as parts other correspondence links which relate the constituent descriptions of `HotelISReq, HotelISDes` and `HotelISImpl.` Note that the correspondences of the constituents need not be one-to-one. The requirements description, for instance, may contain descriptions of activities taking place in the hotel environment within which the information system will function, such as the organization of an event (e.g., a wedding reception) for which there are no corresponding descriptions in the system design. Likewise, several implementation descriptions, say procedures `ChargeExpense` and `GenBill` may correspond to a particular design description, say `TrackExpenses`, which is intended to keep track of all expenses associated with a particular client. Conversely, several design descriptions may correspond to a single implementation description. In the example, the design descriptions `Customer` and `VIPCustomer,` which may be part of a generalization hierarchy for customers, correspond to a single implementation description `Customer` (because, for instance, all customer records are stored on a single database relation). In addition to their parts, correspondence links may have additional associated links which justify and otherwise annotate the correspondence relationship, as done in the DAIDA environment [Jark92].

The SIB model emphasizes the use of carefully controlled correspondence hierarchies through the notion of *application frames* (AFs.) Each application frame includes at least one implementation and optional design and requirements descriptions:

```
Description ApplicationFrame in Metaclass with
     hasParts
               reqDesc : RequirementsDescription
               desDesc : DesignDescription
     hasParts, atLeastOne
               implDesc : ImplementationDescription
end ApplicationFrame
```

An application frame can be either a *generic application frame* (hereafter GAF) or a *specific application frame* (SAF). A SAF describes a particular complete information system and includes exactly one implementation (and optional design and requirements descriptions). A GAF, on the other hand, is an abstraction of a collection of applications pertinent to a particular application domain and includes one requirements description (describing the application), one or more designs and one or more implementations for each of these designs.

```
Description SAF in Metaclass isa ApplicationFrame with
     hasParts, exactlyOne
               implDesc : ImplementationDescription
end SAF

Description GAF in Metaclass isa ApplicationFrame with
     hasParts, exactlyOne
               reqDesc : RequirementsDescription
     hasParts, atLeastOne
          desDesc : DesignDescription
end GAF
```

GAFs can be viewed as idealized design histories of SAFs, their evolution reflecting the accumulation of experience in deriving SAFs from GAFs and developing them further. Application frames can also be specialized according to the application for which an information system is intended, such as Text Processing or Public Administration.

Summarizing, generalization/specialization hierarchies (subtyping) "specialize towards a particular domain", for example, from administration in general to public administration. The upper classes contain less information than the lower ones (typically modeled by adding new attributes to subclasses, or by stronger constraints) These hierarchies concern the "depth" of knowledge covered. Genericity/specificity hierarchies (incremental modification or parameterization) tend to specialize towards a narrower class of solutions, for example, from generic aggregations of requirements, design, and code to specific ones in which parameters are instantiated, or, say, from sorting procedures where the base type to be sorted is a parameter, to ones where the base type is Integer. Finally, correspondence hierarchies tend to specialize towards implementations. Thus, from the point of view of correspondence, an implementation is below a design which, in turn, is below a requirements specification. Vistas along the correspondence hierarchy are called application frames (AF) and they play a major role in the ITHACA methodology.

## 2.4. Similarity

*Similarity* links represent similarity relationships among software objects and provide a foundation for approximate retrieval from the SIB. Similarity has been studied in psychology [Tver77] and AI, most relevantly to this work within the context of case-based reasoning [Barl91]. Similarity has also been offered, within the context of object-oriented systems, as a generalized version of generalization [Wegn87]. Its applications include the support of *approximate* retrieval with respect to a software repository as well as the re-engineering of software systems [Schw91].

In general, similarity is a relation determined by a flexible comparison between distinct constituents of two entities. Quantitatively, the result of the comparison can be interpreted either as a measure of closeness in some abstract space [Tver77, Mich86], or as a probability of the entities resembling each other even when possibly missing constituents are taken into consideration [Russ88, Espo92]. In the present work we adopt the first

interpretation. Thus, similarity links are *derived* links, computed with respect to some abstractions, either explicit (represented in the SIB) or implicit (in the user's mind).

Within the SIB, we are primarily interested in similarity with respect to the abstractions (kinds of links) explicitly defined in it, as only such similarity links can be computed automatically. On the other hand, user-defined similarity links are also allowed for flexibility reasons. Similarity is computed with respect to *similarity criteria* and expressed in terms of corresponding *similarity measures*, which are numbers in the range [0,1]. An aggregate similarity measure with respect to a set of criteria can be obtained as a weighted aggregate function of single-criterion similarity measures, the weights expressing the relative importance of the individual criteria in the set. This measure may be symmetric or directed. For example, similarity with respect to generalization may be defined as symmetric, whereas similarity with respect to type compatibility of the parameters of two C routines may be defined as directed.

Similarity can be used to define task-specific partial orders on the SIB, thus facilitating the search and evaluation of reusable software objects. Moreover, subsets of the SIB can be treated as equivalence classes with respect to a particular symmetric similarity measure, provided all pairs of the class are more similar than a given threshold. Such similarity equivalence classes may span different application domains, thus supporting inter-domain reuse.

## 2.5. Informal and User-Defined Links

Another important link type is that of *derivedFrom* links for version management. As in other models of versions [Katz90], the version space of a description would be structured as a tree with *derivedFrom* links pointing away from the leaves (the latest versions) and towards the root (the initial version).

A key issue in any version model is the propagation of changes from a description to other related ones through correspondence or hasParts links. Configuration management tools such as those described in [Rose91] will be adopted to address this issue. Another planned extension is the inclusion of links denoting procedure calls within implementation descriptions. Such links can be particularly useful in debugging, software modification and reverse engineering.

In general, if users have foreseeable demands for other link types, they can define them through the mechanisms provided by Telos. For unforeseen representational needs, there is a hypertext link type which makes it possible to attach informal annotations or animations to any SIB object.

## 2.6. Associations

The SIB described so far can be viewed as a global information base where everything is visible and accessible through a symbol table which contains external identifiers for particular SIB objects. For example, the simple SIB of Figure 2.3 includes external identifiers `SoftwareObject, Person, Date, VersionNumber, HotelIS, Accounting, Cashier, Bar, 6.03.1, Sept90` and `Yannis`. In general, the SIB will also contain descriptions with no external identifiers or with several.

*Association* is intended to allow the grouping of descriptions that play together a functional role [Brod84]. For example, we may define as an association the descriptions that constitute a design specification for a hotel information system, or all the classes that define an implementation of that same system. Note that association partly addresses the need for encapsulation facilities in conceptual modeling. The contents of an association can only be accessed through the entry points defined in its symbol table.
Thus, an association is, actually, a tuple:

```
Association = (setOfDescriptions, symbolTable)
```

The SIB itself is a global association containing all objects. Its symbol table contains all the external identifiers of every object. Name conflicts are resolved by a precedence rule.

Associations can be combined to define new ones. Assuming that the functions `space` and `symTable` access respectively the set of descriptions and the symbol table of an association, and that the components of entries of the symbol table can be accessed through `identifier` and `range`, we can define

```
association1 = ({X| X ∈space (SIB) and instanceOf(X,  DesignDescription)},
                {Y| Y ∈ symTable(SIB) and range(Y) ∈
space(association1)})
```

or,

```
association2 = (space(SIB) - space(association1),
                {X | X ∈ symTable(SIB) and
                 identifier(X) ∉ identifier(association1)} )
```

Associations can be defined then as special descriptions having the structure:

```
Description Association in OmegaClass with
     attributes
          author: Person;
          importFormula , exportFormula: DerivationFormula
     hasParts
          space: Description;
          symTable: SymbolTable
end Association
```

The `importFormula, exportFormula` components of an association are assumed to be maintained automatically and keep track of interdependencies in the definitions of associations. In the earlier examples, `association1` imports from the SIB and exports to `association2`, while `association2` imports from SIB and `association1`.

Associations can be considered as materialized views, defined through queries, or from other associations through set-theoretic operations. For pragmatic reasons, the SIB offers another form of modularization, called *views*, where the defined groupings are not materialized. Like their database cousins and unlike associations, views cannot be updated directly, but only through updates of their `importsFrom` associations.

## 2.7. The Global SIB Structure

The structure and the meaning of each requirements, design or implementation description depends, of course, on the notation -- linear, graphic or other -- used for that description. To accommodate different notations -- say, SADT or ORM [Pern90] for requirements, E-R diagrams or some object-oriented notation for design, C++ or COBOL for implementation -- requires facilities for modeling the nature of the symbolic structures accommodated by that notation. This is achieved within the SIB through extensive use of the classification dimension.

Figure 2.5 shows a number of simple C++ class descriptions, including the class GenBill, which together define the implementation of a hotel information system, say `HotelISImpl`. The latter is an association and is part of an application frame named `HotelIS`. All of the above are simple classes within the SIB. The figure also shows some of the metaclasses that might be instantiated during the process of inserting such application frames in the SIB. In particular, the figure shows the metaclasses `C++Class, C++Method, C++Object` whose instantiations populate C++ implementation descriptions such as `HotelISImpl`. All C++ objects are included in the `C++ association` which is an instance of the metametaclass `ImplModel`, along with, say Smalltalk and CooL. SADT, on the other hand, is listed under `ReqModel`. Some models, the Entity-Relationship model for example, may be listed under more than one metametaclass model.

The picture of the SIB structure suggested by Figure 2.5 can now be augmented with that of Figure 2.6. The token level of the SIB is reserved for information concerning run-time experiences with software described in the SIB. For example, the `HotelISImpl` association is shown in Figure 2.5 to have been run for `Lato` (presumably, a hotel) with attached information on performance characteristics and bug reports for classes included in the association, such as `GenBill`. The simple class level of the SIB includes program descriptions, mostly declarations, along with descriptions of the designs and requirements from which they were derived. In addition, the simple class level includes more macroscopic units such as associations representing implementation, design or requirements descriptions and application frames. Finally, the metaclass level includes generic descriptions of application frames as well as requirements, design and implementation descriptions.

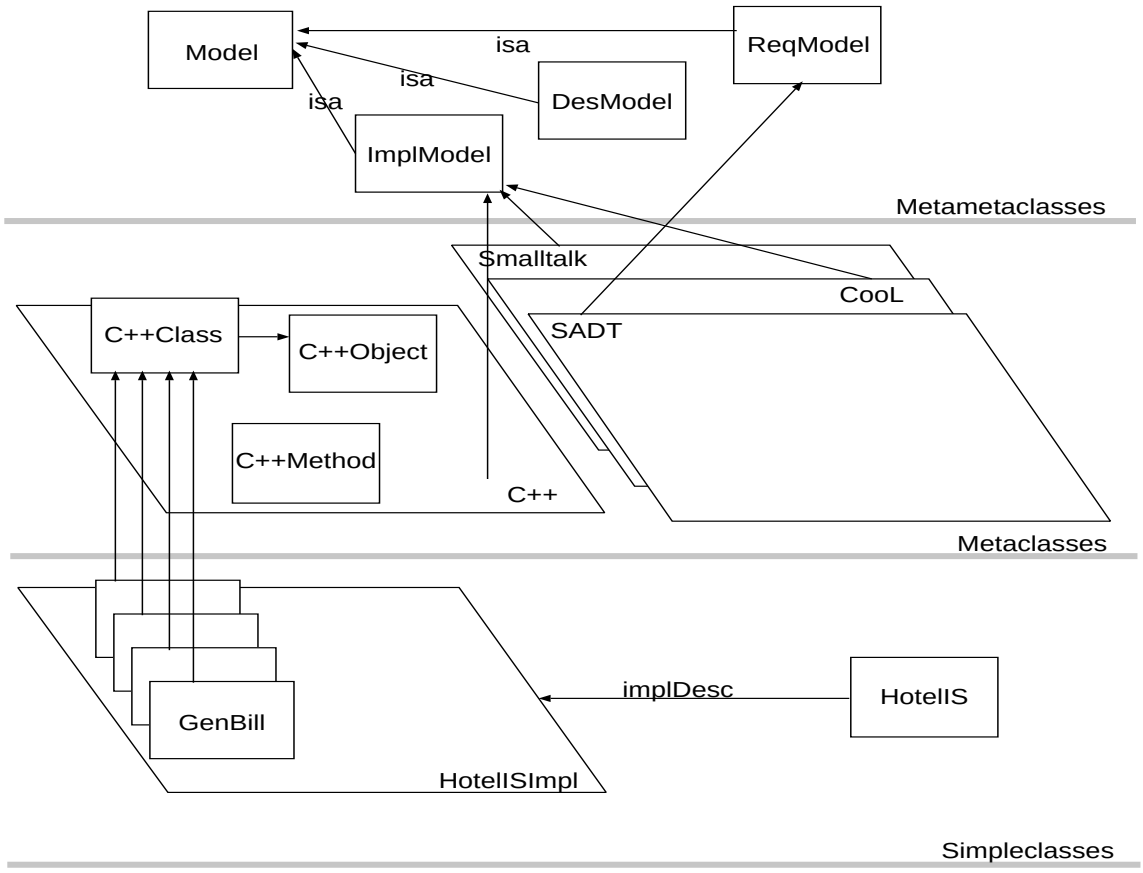


Figure 2.5: Global SIB Structure

As mentioned earlier, all SIB objects are subject to modification, not just the tokens. However, pragmatic reasons dictate the adoption of different operational rules for objects at different classification levels. In particular, modifications at meta-levels are relatively rare and under the authority of designated engineers, while at the simple class level application developers actually change the schema by populating the SIB with software descriptions (either manually or through development tools).

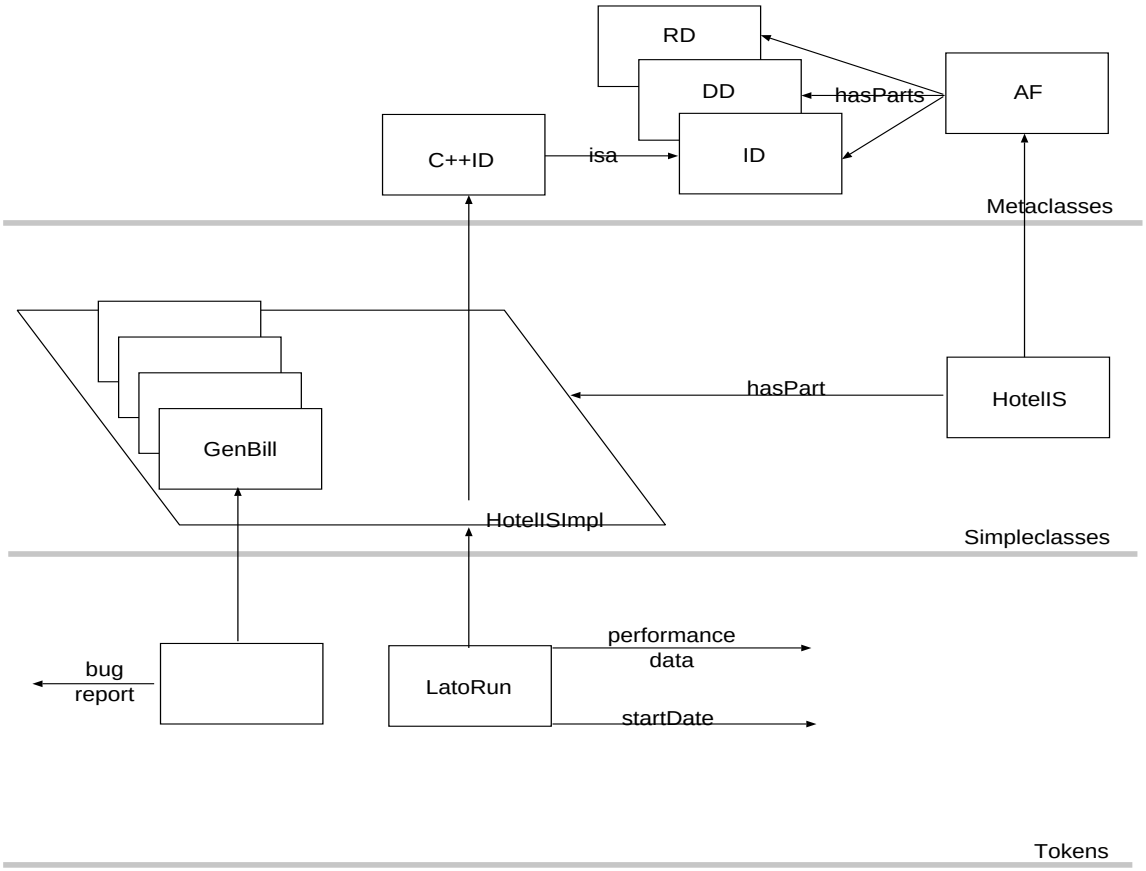


Figure 2.6: SIB Structure and Application Frames

# 3. The SIB Prototype System

This section describes an industrial strength prototype SIB system that has been implemented at the Institute of Computer Science, FORTH. This robust prototype has been functional for some time and has been made available to other sites for experimentation. After an overview, the section presents the system's architecture and user interface, followed by a discussion of its implementation and integration with other tools within the ITHACA application development environment. A usage scenario is detailed in Section 4 as part of the SIB evaluation.

## 3.1 System Functionality

The SIB system offers a number of maintenance, selection and workspace management functions. Maintenance functions include insertion, deletion and update of information in the SIB and are supported by appropriate access language and interactive form- based data entry tools. The latter offer a form based on the type of object to be edited by reading out schema information from the SIB itself. Selection functions include querying and browsing, while workspace management involves the dynamic definition and modification of workspaces to provide easier and more efficient interaction with the SIB.

Maintenance and selection operations are performed on *workspaces* which are application-specific and/or user- specific subsets of the SIB represented as associations. When there are several overlapping workspaces, the identity of shared objects is maintained through reference to the identifiers used within the SIB itself. Classification and generalization links are also shared between workspaces, being an integral part of an object's identity. The use of workspaces offers several benefits, such as focusing attention to selected parts of the SIB by hiding irrelevant information; creating convenient views of the same objects either by renaming them or by hiding particular parts of their descriptions; improving search performance by effectively restricting the search space; and supporting privacy. In terms of the structure introduced in the previous section, workspaces are special cases of associations. The default workspace is the entire SIB (which can be thought as a global workspace).

Queries to the SIB can be classified from a user's point of view as *explicit* or *implicit*. An *explicit* query involves an arbitrary predicate explicitly formulated in a query language or through an appropriate form interface. An *implicit* query, on the other hand, is generated through navigational commands in the browsing mode, or through a button or menu option, for frequently used, "pre-canned" queries. Browsing commands and explicit queries can also be issued through appropriate interfaces from external tools.

The selection of software descriptions from the SIB is accomplished through the *Selection Tool* (ST) in terms of an iterative process consisting of querying and browsing. Browsing is usually the final and sometimes the only step required for selection. The functional difference between the retrieval and the browsing mode is that the former supports the retrieval of an arbitrary subset of the SIB and presumes some knowledge of the SIB contents, while the latter supports local exploratory searches within a given subset of the SIB without any prior knowledge. Operationally, both selection modes evaluate queries against the SIB.

Browsing in a software development environment is a powerful and required facility, as evidenced from the emphasis on good browsers in almost all available software class libraries (see also, [KoMc92]). Of course, when offered as the only access mechanism in large libraries, browsing has its limitations. On the other hand, when augmented with filters and orientation facilities, and coupled with additional retrieval modes, browsing becomes a very effective access mode.

The relevance of a software description to some application, the similarity of two descriptions with respect to some criterion, the coupling of two pieces of code in a running system, are but a few examples of non-Boolean predicates concerning software. The SIB system is intended to support such non-Boolean queries through tools that order their response by relevance, similarity, affinity, and the like.

The basic functions of the SIB are as follows:

*Maintenance functions:*
*Insert:* Descriptions X Associations -> Associations
*Delete:* Identifiers X Associations -> Associations
*Update:* Descriptions X Associations -> Associations
*NewVersion:* Descriptions X Associations -> Associations

The *Insert* function takes as input a description and an association and inserts that description to the association as well as the global association (SIB). If the description is that of an association, insertion includes materialization of the association. *Delete* takes as argument a description and an association and deletes the description from the association. *Update* modifies a particular version of a description, while *NewVersion* turns the updated description into a new version.

*Selection functions:*
*Retrieve:* Queries X Associations -> SetOf (Descriptions X Weights)
*Browse:* Identifiers X SetOf (Links X Depths) X Associations -> Views

The *Retrieve* function takes as input an association and a (compound, in general non-Boolean) query and returns a subset of the association with weights attached indicating the degree to which each descriptions in the answer set matches the query. The prototype implementation only handles Boolean queries, but extensions to handle non-Boolean ones are already underway. Queries are formulated in terms of the query primitives offered by the Programmatic Query Interface. A set of queries of particular significance can be pre-formulated and offered as menu options, thus providing maximum ease-of-use and efficiency for frequent retrieval operations.

Browsing clearly is a special retrieval operation which begins with a particular SIB description which is the current focus of attention (called the *current object*) and produces a view of a *neighborhood* of the current object within a given association. Since

the SIB has a network structure, the neighborhood of the current object is defined in terms of incoming and outgoing links of interest. Moreover, the size of the neighborhood can also be controlled. Thus, the *Browse* function takes as input the identifier (name) of the current object, a list of names of link classes paired with depth control parameter values and an association, and determines a local view centered around the current object.

When the depth control parameters are all equal to 1, a *star view* results showing the current object at the center surrounded by objects it is connected through links of the types selected. This is the simplest and smallest neighborhood of an object, in topological terms, with a controllable population. *Browse* can be called iteratively with argument one of the objects contained in the browser's view, resulting in a new current object and an updated view. Effectively, the *Browse* function provides a moving window with controllable filters and size, which allows navigational search over subsets of the SIB network.

When the depth control parameters are assigned values greater than 1, the *Browse* displays all objects connected to the current object via paths consisting of links of the selected types (possibly mixed), where each type of link appears in a path up to a number of times specified by the corresponding depth parameter. This results in a directed graph rooted at the current object. Finally, when the depth parameters are assigned the value ALL (infinite), the transitive closure of one or more link types is displayed with respect to the current object. Such a browse operation can display, for example, the call tree of a given routine.

The multimedia nature of SIB descriptions calls for the development of a hypermedia annotation mechanism that would gracefully complement the SIB semantic network. This is accomplished by establishing referential links between descriptions, treated as a special category of attribute links, thereby integrating them within the SIB network model. Hypermedia annotations include text, graphics, raster images and algorithm animations.

## 3.2 The SIB Architecture

The SIB system consists of the following modules (Figure 3.1):

- The *Interactive User Interface* generates and coordinates the other parts, including the interface tools of the Data Entry Forms and the Selection Tool. It is implemented using the OSF/Motif toolkit.
- As a component of the *Selection Tool*, the *Graphical Browser* presents parts of the SIB network graphically and allows the user to browse through it by sending messages to the Interactive User Interface in response to user actions. The Graphical Browser is a LABY[2] graphical editor with only the working area present.
- The *Data Entry and Display Forms* offer form interfaces for presenting and entering data. The display forms are used to provide information about the current object or

[2] LABY is a general purpose graphical editor developed in part within the ITHACA project at the Institute of Computer Science, Crete (FORTH) [Kate90].

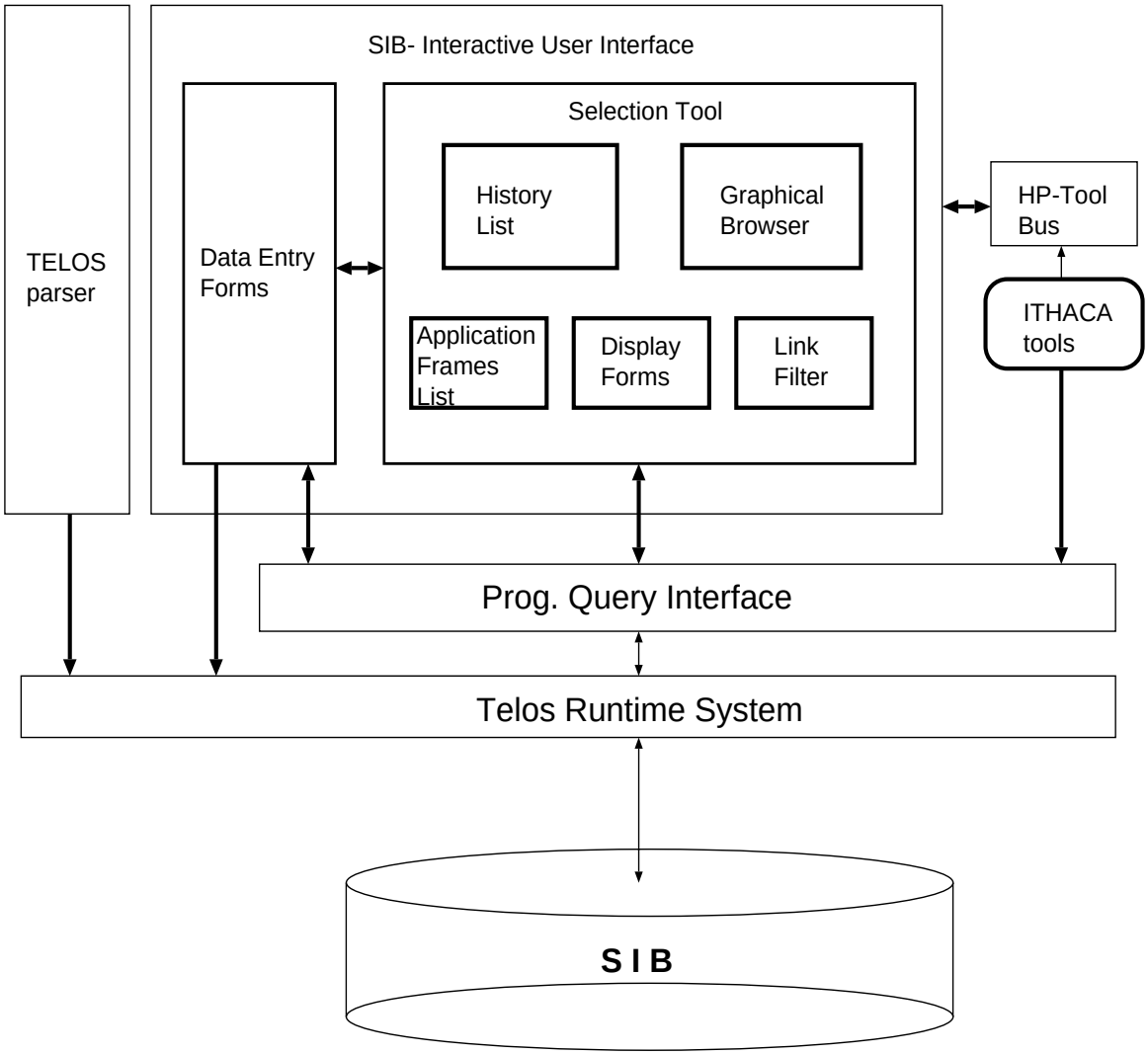


Figure 3.1. The SIB architecture

another selected node in a form layout. The forms have been designed to support multimedia information (text, graphics, images, animation, etc.)

- The *Programmatic Query Interface* handles queries issued by various components of the Selection Tool, the Data Entry Form, or external tools. Processing of a query results in the construction of a file suitable for display by the Graphical Browser or the Data Entry Form. In the current implementation a separate query interface (based on the client-server model) is used for programmatic queries. However, the two query interfaces will be integrated in the future.
- The *ITHACA Tools*, integrated with the SIB at command level, use explicitly the Selection Tool for retrieval or access directly the programmatic query interface for query and data entry operations.

- The *Telos Runtime System*, described in Section 3.4, constitutes the kernel of the SIB system.

## 3.3 User Interface

The user interface of the system consists of the following windows (Figure 3.2):

- *The Graphical Browser*:

The Graphical Browser, built using the LABY graphical editor, displays a part of the SIB network in the neighborhood of the current object. The window of the Graphical Browser is topologically divided in two parts. From each node appearing in the lower part emanates at least one link pointing to the current node. Likewise, there is at least one link emanating from the current node and pointing to each node appearing in the upper part. The types of links are represented by a color code which is shown in the Link Filter. Nodes appearing in the graph of the browser are selectable with the mouse. The links displayed at any one time include direct links from the current object to/from other objects and computed isA and instanceOf links.

The population of the display is controlled by means of the *Link Filter* (see below) and the *Instance Box*. The Instance Box appears in the display of the Graphical Browser when the instances of a certain active link type and adjacent to the current object are too many to be shown on the display. On selecting the Instance Box of a link class with the mouse, a list of objects related to the current one by that type of links appears. The objects on this list are selectable just like those displayed graphically.

- *The Link Filter*:

The Link Filter provides buttons corresponding to link types and is used for activating/deactivating links thus controlling the information displayed in the Graphical Browser. The isA and instanceOf buttons further offer the option of displaying inherited as well as direct isA and instanceOf links. All buttons show the color code of the link classes and come with a help facility.

- *The History List*:

The History List is a navigation aid intended to prevent users from getting lost, a common problem in hypertext systems [Conk87]. The History List is scrollable and contains the names of the objects selected as current during a session in chronological order, the most recent one shown at the bottom (as with the history command of Unix). All entries of the list are selectable. A selection made on the History List is functionally equivalent to one made on the Graphical Browser.

- *The Application Frames List*:

The Application Frames List contains the names of all application frames, thus presenting a bird's-eye-view of the SIB. This facility is available as compensation for the limited scope of the Graphical Browser, clearly a shortcoming for broad searches. The application frames are displayed in an indented list representing their hierarchical structure. Each item on the list is selectable, which effectively allows big jumps within the SIB network while in the browsing mode. Moreover, the Application Frames List serves as the initial entry point to the Selection Tool.

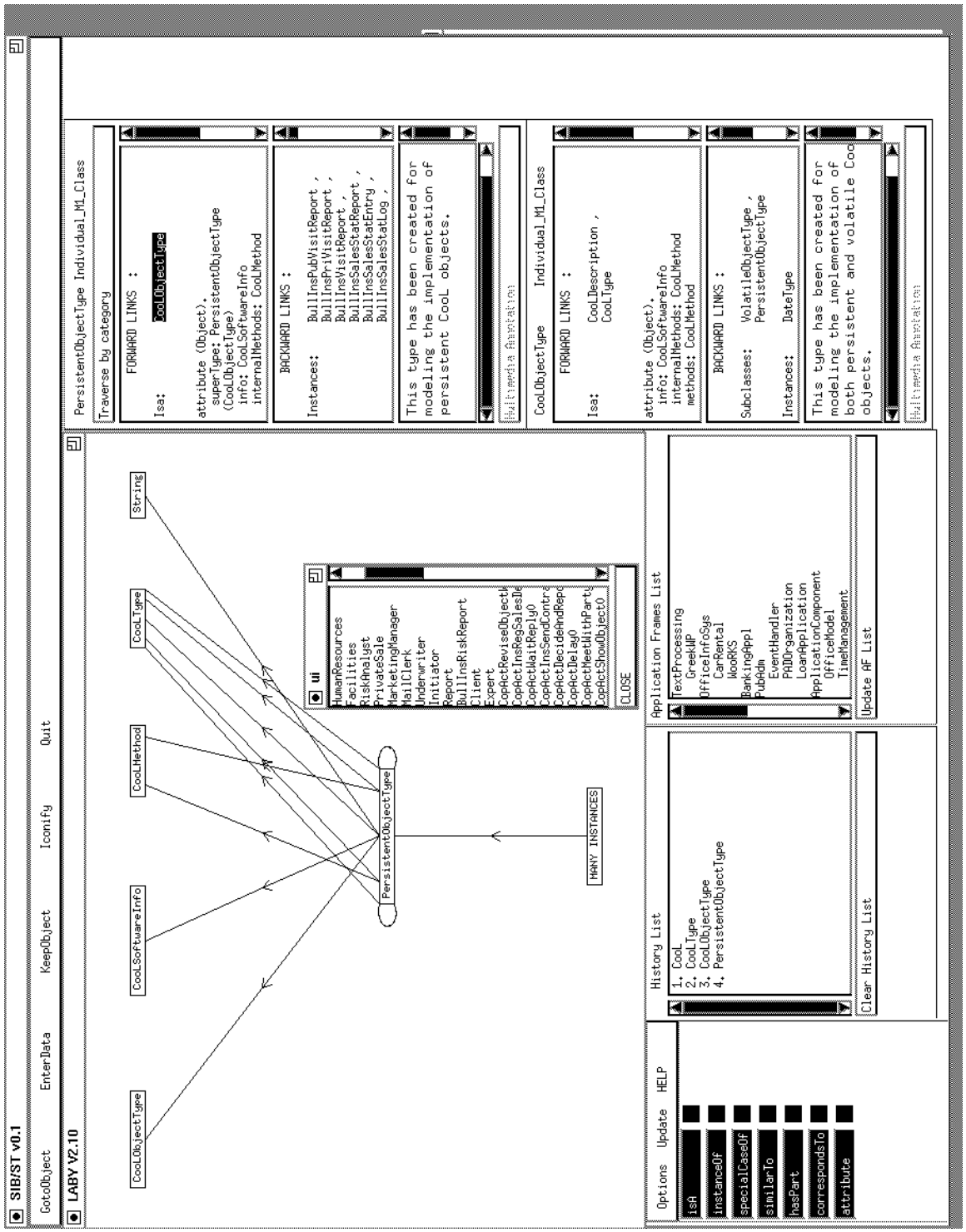


Figure 3.2: The SIB user interface.

- *The Main Form*:

The Main Form is the top right-most window of the Selection Tool and invariably displays information about the current object. This information includes the abstracted definition of the object in a form layout, generated by unparsing the SIB descripotion of the object. The form may also contain multimedia annotations for the object, each one displayed in a separate window. At present, this annotation is textual and graphical, but can be of any other type with no additional effort, provided that appropriate tools exist within the SIB's operating environment.

- *Auxiliary Form*:

This form is used to display information about an object other than the current one, without changing the actual view in the Graphical Browser (see Figure 3.2). To minimize user distraction, only one auxiliary form can be open at a time and objects are not selectable on auxiliary forms. The auxiliary form is actually a preview mechanism and is offered as an orientation aid.

- *The Button Panel*:

Through the Button Panel, several other windows for performing a variety of useful functions can appear on demand. Currently these functions are:

*GotoObject*: Allows direct access to invisible objects by name.

*EnterData*: A Data Entry Form is offered for entering data into the SIB.

*KeepObject*: Keeps a retrieved object in a local workspace.

*Iconify* and *Quit*: Closes and iconifies a window; and quits the Selection Tool respectively.

## 3.4 Implementation Aspects

As indicated earlier, the SIB model is based on Telos without its temporal reasoning mechanism and its assertional sublanguage. Early experimentation with traditional and object-oriented database systems proved Telos to be definitely superior in modeling flexibility. Moreover, its SIB implementation is based on C++ and clearly outperforms the earlier Prolog implementation of the language.

Before proceeding with an overview of the system implementation and the choices we have made, we list some of the object management and modeling requirements which actually shaped the implementation.

Foremost among these requirements is the need for a powerful and expressively rich data model. This requirement effectively ruled out traditional DBMSs based on the classical data models. However, there have been noteworthy investigations of the modeling requirements of software applications, including the studies of versioning models summarized in [Katz90], the mechanisms supported in CACTIS [HuKi89] for derived data and the configuration management model of [Rose91]. Within Software Engineering, some work has been done on the identification of relationship types for describing software objects [Meye85]. Along a different direction, terminological languages such as CLASSIC [Borg89] offer facilities for defining terms which include a *subsumption* operation that determines automatically whether one term description is more specialized

than another. LaSSIE [Deva91] illustrates graphically how such a facility can be used for a software repository.

A second requirement is the need for effective and efficient support for concurrent usage of the SIB, in addition to adequate query processing facilities. Unlike databases, software information bases contain mostly schema descriptions. Moreover, these schema descriptions include cycles and evolve dynamically. These considerations rule out the direct use of standard database implementation techniques, such as two-phase locking or directed acyclic graph methods for concurrency control. Current research indicates that the features of database transactions (Atomicity, Concurrency, Isolation, Durability) will have to be separated into multiple services, where each of these services might look quite different from the one used in traditional DBMS. Software designers do not want to work in isolation but in overlapping workspaces with explicit communication. Atomicity and recovery have to be separated since no one wants to reset a whole design transaction just because a conflict occurred. Likewise, the rich structure of software information bases such as the SIB render traditional query optimizations for DBMSs ineffective.

Of course, there has been useful research within databases which can readily be adopted to improve the efficiency and safety of software information bases. For example, extending traditional database types with long fields and complex object structures eliminates the need for costly file-opening and closing operations when scanning software information and means that common subparts can be shared among complex objects in a controlled manner. The DAMOKLES system [Ditt87] is an excellent example of this kind of extension.

A final requirement on the SIB calls for a a host of functions not available in DBMSs including configuration managers which support the efficient and consistent reconfiguration of complex objects when components have changed (as in CACTIS [HuKi89], for example), and similarity measuring tools (as in [Schw91], for code descriptions).

In summary, current database technology leaves much to be desired in terms of supporting software information bases intended for reuse. Accordingly, our work has been founded on a richer semantic data model which can be thought as a layer on top of emerging object-oriented database systems. Nevertheless, the implementation technology for DBMSs in general and object-oriented database systems in particular has served as source of ideas and inspiration throughout. Admittedly, it is still an open question whether such a layered approach will be feasible for very large software information bases, or whether database technology will advance to satisfy some or all of the requirements discussed here.

As indicated already, the SIB has been developed as part of a complete application development environment containing several tools. Their interconnections are shown in Figure 3.3 while Figure 3.4 presents the Telos runtime system.

Architecturally, the prototype SIB system contains all the main components of a typical object-oriented DBMS implementation, but low-level optimizations and data structures are rather different [Dado92,Cons93]. All data fields are set-valued and there has been much emphasis on efficient handling of network-like structures and fast retrieval of transitive closures for particular link types. All links can be traversed bi-directionally and there is direct support for all atomic retrieval operations (e.g., find all classes an object is

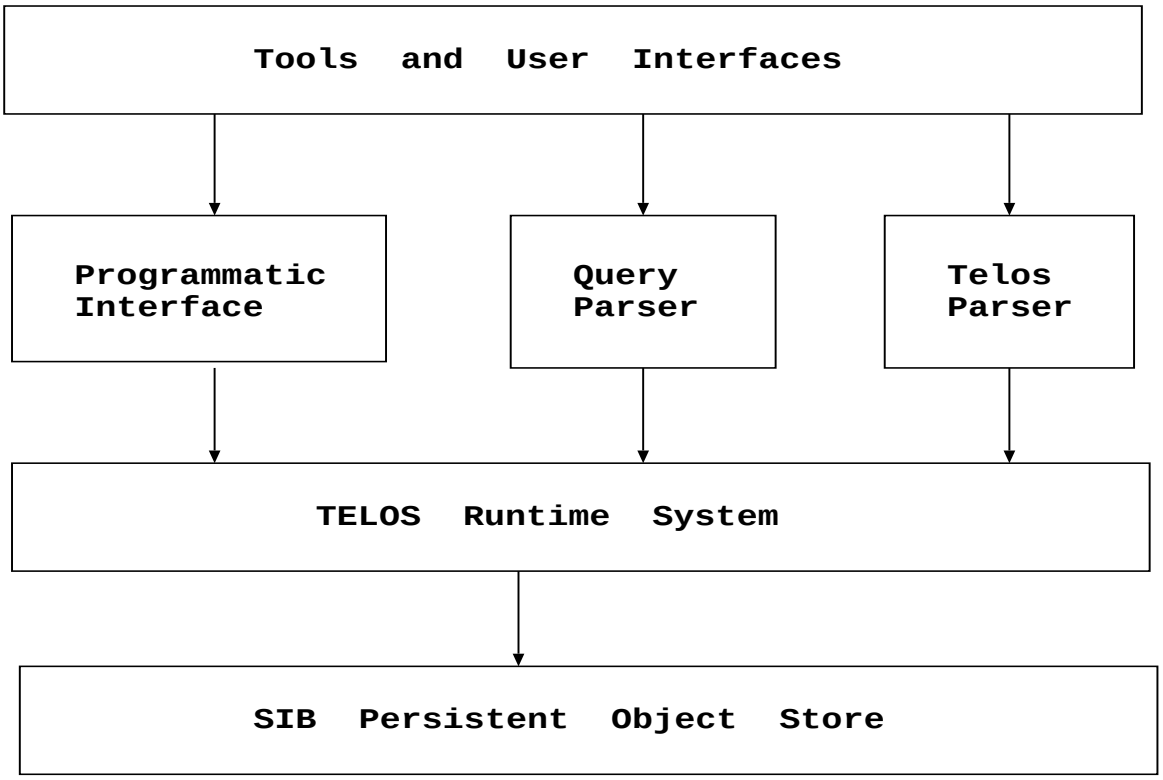


Figure 3.3: Interfaces of the SIB with other tools and the object store

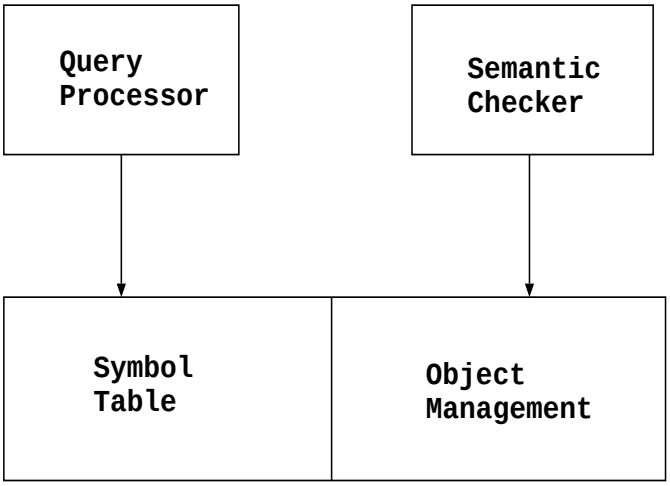


Figure 3.4: Structure of the Telos runtime system

an instance of, find all related objects, etc.). No optimization is done for range queries on primitive values (integers, floats etc.). The schema is maintained completely at data-level, allowing for fast schema extensions at run-time. The data of a software information system are in general rather static, with infrequent and bulk changes, suggesting a strong preference for query optimization over update optimization. Telos objects are presented in terms of their (hopefully meaningful) external identifiers. Like object-oriented databases, objects have associated unique and system-supported internal identifiers [KhCo86, WKDa89], which are invisible to the user.

Five C++ classes are used to represent all Telos objects. The contents of the five classes are sets of system identifiers for the *instanceOf*, *isA*, *attribute* and *trigger* relations respectively. All these relations are kept in both directions to allow fast query processing. Triggers are either built-in or user-supplied, in which case they must be linked to the runtime system. Triggers can be used to provide access to data outside the database, thereby offering a powerful interface mechanism.

The Telos objects are kept persistent on disk and copied to memory in a cache-like manner. The system implemention employs demand loading in the current prototype, to be replaced later with a prefetch mechanism [Low88]. During the execution of a transaction, all modifications done on existing objects and newly created ones are kept in memory and only become persistent at the end of the transaction. Queries may be allowed on these objects before the end of a transaction under certain restrictions. All dynamic memory allocations are done on a fixed granularity base, first to reduce allocation time and, more importantly, to avoid cluttering the virtual address space after longer execution times. Similar implementation issues have been discussed in [KhFr88, MeLa87, SkZd86]. The association of system identifiers with object locations on persistent store and/or in memory is done by the system catalogue. The system identifiers are kept dense in the sense that the numerically lowest free identifier is allocated first. Moreover, since system identifiers are only internal, there are no identity conflicts. Identifier density allows a virtual memory-like indexing of the system catalogue with use of page tables. Since it requires an additional indirection step after a growth of the database by a factor of 1000, this design leads to nearly size-independent performance. Symbol tables translate from external identifiers to system identifiers and vice versa. The symbol table tuples are organized as the system catalogue, giving good performance for translations from system identifiers to external identifiers translation. The inverse translation, however, is supported by B-trees and is the only component whose performance degrades with growing database size. Fortunately, the frequency of the inverse translation operation is several orders of magnitude lower to forward translations, since queries usually return larger answer sets than their argument set and all internal query processing is done in terms of system identifiers. Symbol tables as well as the system catalogue are cached.

Tests for measuring the performance of the SIB with a population of 12,000 objects (links and nodes) yield query response times between 1 and 4 seconds (the maximum occuring for recursive queries following over 1,000 links). With a population ten times larger (120,000 objects), the response times for the same queries (now following up to 10,000 links) range from 1.2 to 4.5 seconds.

Concurrency control in network-like structures is still an open problem. Database parts to be locked can be hardly determined. Based on the assumption that updates are not too

frequent, we have built into the implementation only read and write locks for the whole database. This implies that locks may be held only for short time intervals and that interactive data entry form operations must check consistency before they commit their data to the database. Cache invalidation is done at lock grant on demand of each server instance, thereby minimizing degradation of performance with an increasing number of clients. The lock mechanism works reliably on local area networks.

Finally, the implementation has emphasized *portability* of the platform across all UNIX systems and a variety of hardware settings, including PC-based ones. The system currently runs on Sun3, Sun4 series, SparcStations and 386 machines under UNIX. The X window system is required, preferrably with a color monitor. The system may run in a local area network being based on a client server architecture. Query processing under use of caches of controllable size is mainly done on the server side. Usage of shared caches for read-only access is currently under investigation.

# 4.
# Empirical Evaluation of the SIB Concept

The SIB model and system have been evaluated in the context of a specific reuse-oriented methodology developed in the project ITHACA. Moreover, we have begun the evaluation of the model with other object-oriented analysis and design methodologies [FiKe92] starting with the Booch design methodology [Booc91]. An obvious advantage of doing the evaluation using the ITHACA methodology is that there already exist substantial amounts of data, generated by companies participating in the project. After all, it is hard to establish the strengths of the SIB unless it is first properly and heavily populated.

This section reviews the reuse-oriented development methodology, the test application domain and applications used to validate the SIB concept, and an extended example from these test applications intended to demonstrate the interaction with the system.

## 4.1 The ITHACA Object-Oriented Methodology

Consider a concrete scenario for software reuse, adopted from [Ader90, deAn91, Fugi92]. The scenario assumes that all software information is organized in terms of Application Frames (AFs) which comprise descriptions of requirements, designs, implementations, and their interdependencies. Thus, an AF provides three views of a software system, plus some process information. As indicated earlier, several different models can be supported for each of the three views. For example, in ORM (Object-Role-Model) [Pern90], the requirements view is a network of application objects connected by roles. Under certain applicability conditions, design-level software object specification can be related to the set of roles the object is intended to support. Similarly, implementation objects are related to their design-level counterparts under certain applicability conditions. In general, design objects may be associated with multiple requirements, and implementation objects with multiple designs, and vice versa.

Development proceeds from a library of Generic Application Frames (GAFs) and the result of a development process is a Specific Application Frame (SAF). From the viewpoint of some application domain, a GAF is an abstraction of a collection of applications pertinent to this domain while a SAF is a specific running application in the same domain. It is assumed that the initial class libraries and an Application Frames library have been generated by *application engineers* and are clustered by application domains. The test application domain is Public Administration, including information systems for office applications. The *application developers* follow the steps in Figure 4.1 to produce a new application [Ader90].

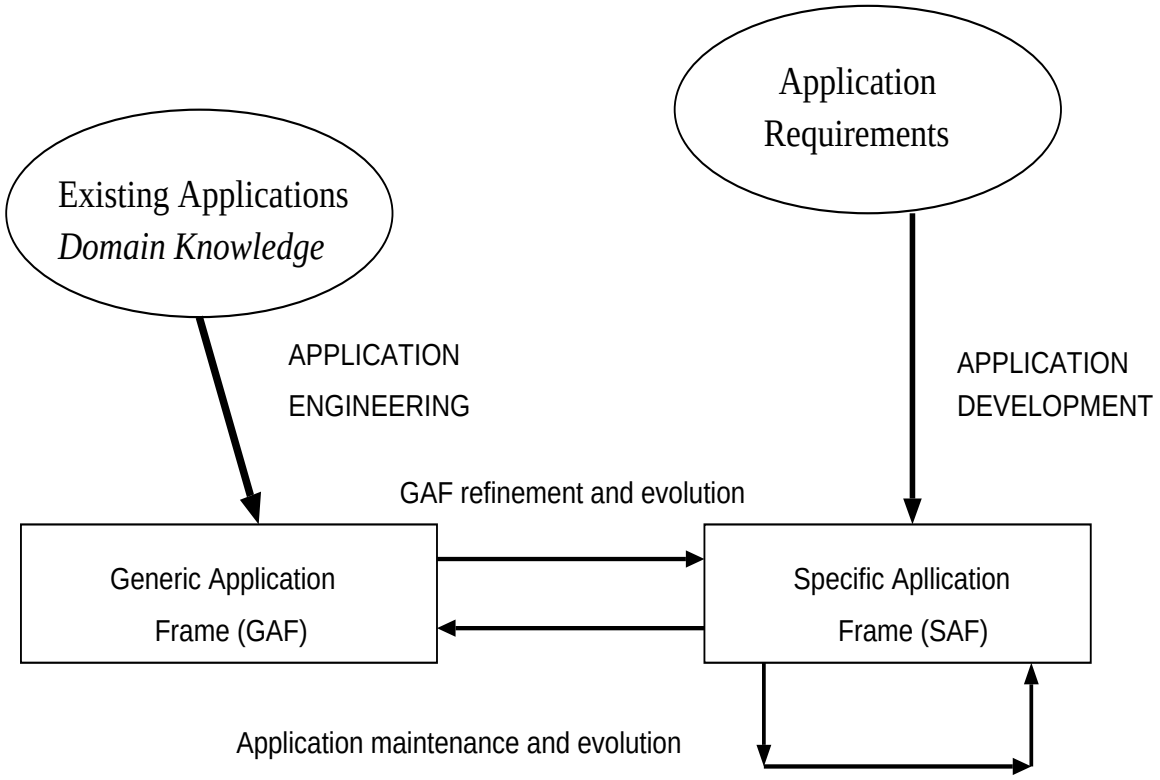


Figure 4.1: Application Development Scenario in ITHACA

1. *Select an Application Frame* from the repository, meeting the requirements for the application being developed.
2. *Select Useful Classes*. In employing an application frame, the developer is guided for the selection of reusable classes from the repository in that the application frame drives requirements collection and specification according to pre- existing generic specifications and designs.
3. *Tailor Classes*. The selected classes are adapted (incrementally modified), using previous design experiences, by supplying parameters or modifying class behavior through inheritance.
4. *Script Application*. A new application is composed by linking design classes together by means of a "script". The script artifacts are to be entered in the repository for future reuse.
5. *Monitor Behavior and continuously develop*. Through testing and validation, or because of changes in the requirements, the application is adapted.

This methodology has been tested with an application development environment that includes the SIB, a requirements collection and specification tool (RECAST [Fugi92], supporting the ORM), a visual scripting tool (VISTA [deMe91]), and the runtime environments of the object-oriented languages CooL [Cool90] and C++.

Among the models that the SIB stores and supports, the ORM, C++ and CooL models are particularly relevant. A comprehensive account of how these models have been stored in the SIB as part of its schema is presented in [Char92]. It is noted that since, the SIB descriptions for the object-oriented programming language models (CooL and C++) are rather straightforward representations of the code, it was possible to construct tools that generate Telos descriptions from CooL and C++ code automatically and at the same time classify them within the SIB.

Public administration has been adopted as the testbed application area for the population of the SIB. Professional application developers from three commercial organizations generated the code and the description of a generic office form workflow system (WooRKS [Ader91]). The system offers assistance to a group of users collaborating to achieve a sequence of tasks in order to accomplish an office procedure. WooRKS consists of models for organizations, information handled, time, operation and coordination of activities. TAO is a specialized application frame (SAF) of WooRKS which supports the organization of events in the public administration domain.

The current population of the SIB consists of about 20,000 separate objects (logical identifiers) for the above application. Most were introduced manually by the professional application developers, since the automatic population tools were not available at the time. The positive usage experiences of the developers are presented in [Proe92] and [Char92]. Most of the interaction of the application developers is through the form-based SIB interface which provides full transparency from Telos. Application engineers, who also use Telos directly, have found the E-R nature of the language and the graphical visualisation very helpful in using it effectively.

## 4.2 An SIB Usage Example

Suppose we are assigned the task of creating an application dealing with processing of letters, intended to assist secretaries, managers and others in writing, checking and mailing professional letters. Looking at the Application Frames List, we observe that there is already an Office Information System called `WooRKS`. The basic concepts in `WooRKS` are actors, roles and procedures.

Our starting point then in exploring the SIB will be `WooRKS`, which we select through the Application Frames List. As we can read in the natural language description seen in the Main Form, `WooRKS` is a work flow system for offices, which handles a variety of activities (see Figure 4.2). Accordingly, it is a reasonable candidate to search for a letter processing application. The description of `WooRKS` includes three attributes which are further explained in its Main Form. One of them, *reqDescr*, deals with `WooRKS` requirements descriptions and it will probably provide us with more information about what the `WooRKS` system actually does. Before making it current, however, we preview its contents on the Auxiliary Form, by selecting `WooRKS1_RD_FORM` from the Main Form, and we decide to visit it.

In Figure 4.3 `WooRKS1_RD_FORM` is current in the Graphical Browser window. Among the classes of activities that it handles, `OrderProcessing` appears to be the most relevant, so we visit it. After we conclude that this is not useful, we return to `WooRKS1_RD_FORM` through the History List and try out `WarehouseProcessing`

and AccountProcessing. which also turn out to be irrelevant to our task . However, we notice that all three of these are instances of FormProcessClass.

Now we take a closer look on FormProcessClass by moving to it through the GotoObject facility (Figure 4.4). As FormProcessClass is a subclass of FormClass, it inherits its attributes. By previewing FormClass, we find out that it has two attributes, *roles* and *baseRole* (see Figure 4.4) and decide to create a new instance of FormProcessClass, called LetterProcessing, whose *roles* will correspond to the initial requirements imposed on our letter processing application. In particular, the *baseRole* of LetterProcessing will be LP_base_role, and the *roles* will be LP_letterCompose, LP_letterCheck, LP_letterApprove, LP_letterSend, LP_letterReceive, and LP_letterArchive.

We have chosen this convention for naming the *roles* by analogy to the *roles* of other, existing activities. To find these names we first made FormRole current using the GotoObject facility (see Figure 4.5). Since FormRole has too many instances to be displayed on the Graphical Browser, an Instance Box appears by clicking on the "MANY INSTANCES" box of the Browser.

At this point we start creating the *roles* and *baseRole* of LetterProcessing. Before creating LP_letterArchive, we visit OP_orderArchive by selecting it from the Instance Box (Figure 4.5). This step turns out to be useful because we find there a *correspondsTo* link from OP_orderArchive to ArchiveAct, which is an instance of ADMActivity (see the Main Form in Figure 4.6). Knowing that ADMActivity handles design descriptions, we proceed to define the ADMActivity corresponding to the new LP_letterArchive in an analogous fashion (we may even reuse the ArchiveAct as the ADMActivity of LP_letterArchive. Similarly, we may choose to use CompileRefAct and/or EvaluationOrderAct which *correspondTo* OP_orderCheck as the ADMActivity of LP_letterCheck, etc.

Finally, we are in a position to define the new LetterProcessing description. We move to FormProcessClass using the GotoObject option and use the EnterData facility, which does not presuppose any knowledge of the Telos syntax. The Data Entry Form for LetterProcessing is shown in Figure 4.7. In the same figure you can see the results of entering the information.

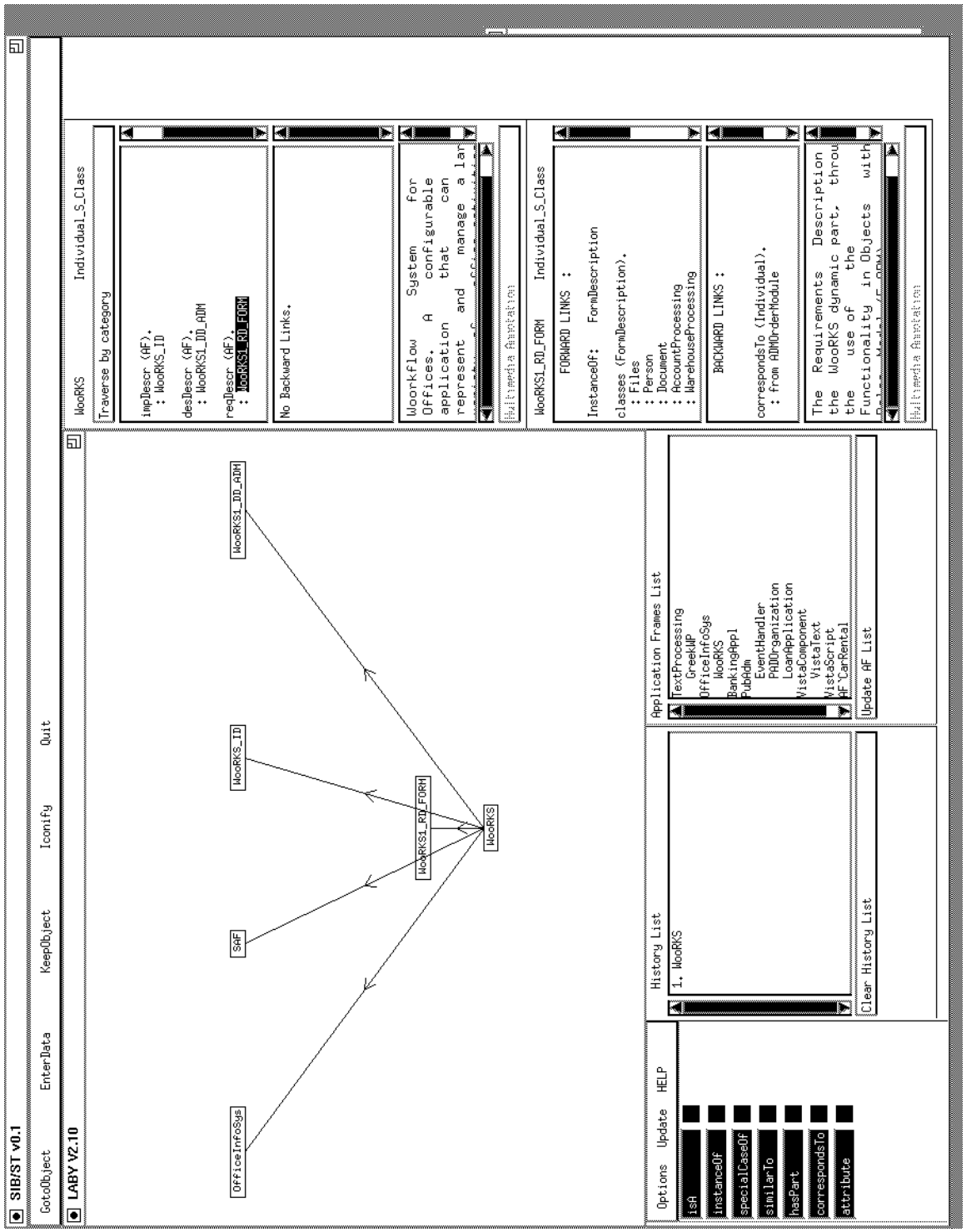


Figure 4.2: The Selection Tool of the Software Information Base. Current in the Graphical Browser is WooRKS, the starting point of the usage example.

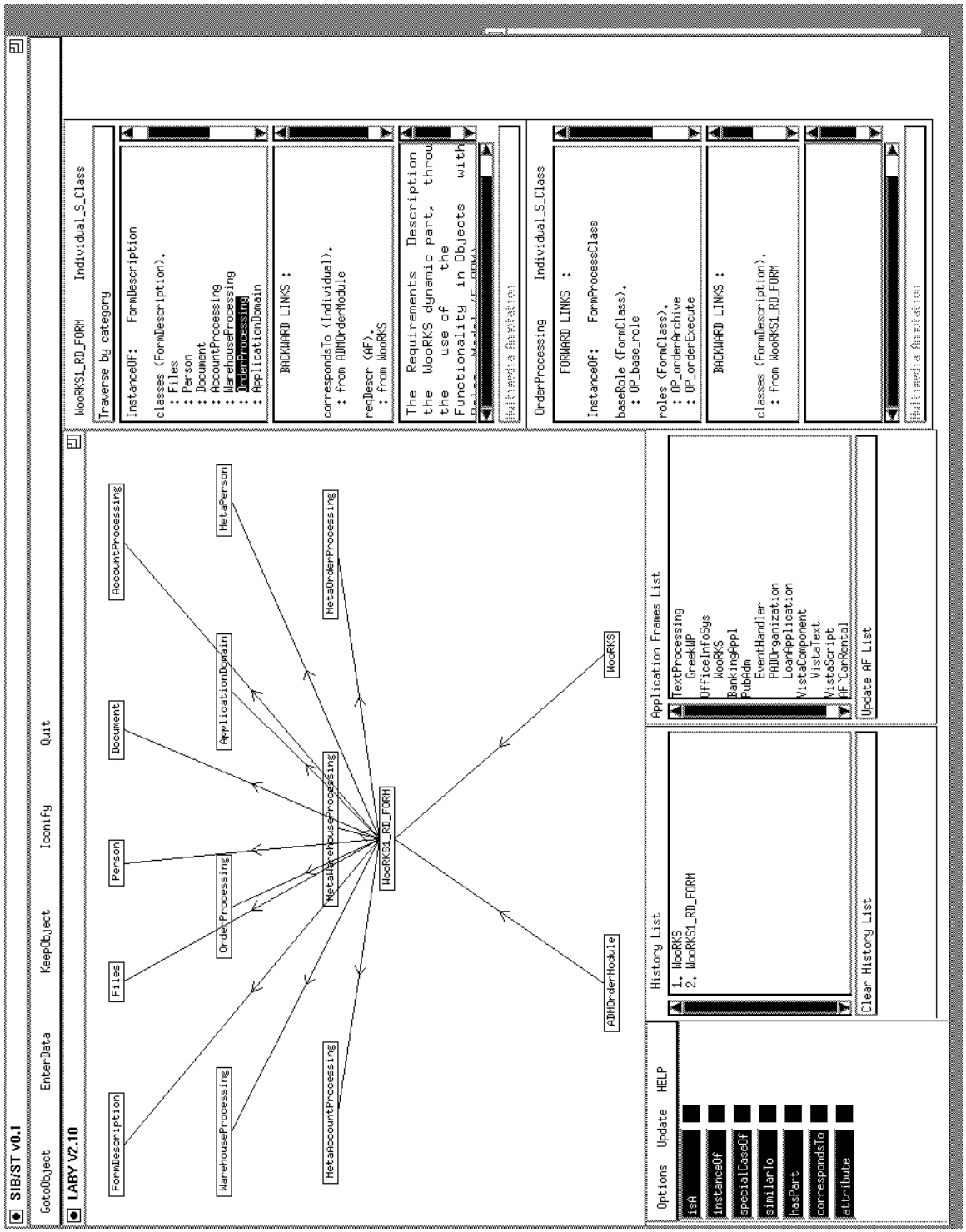


Figure 4.3: Inspecting the requirements of WooRKS and previewing OrderProcessing.

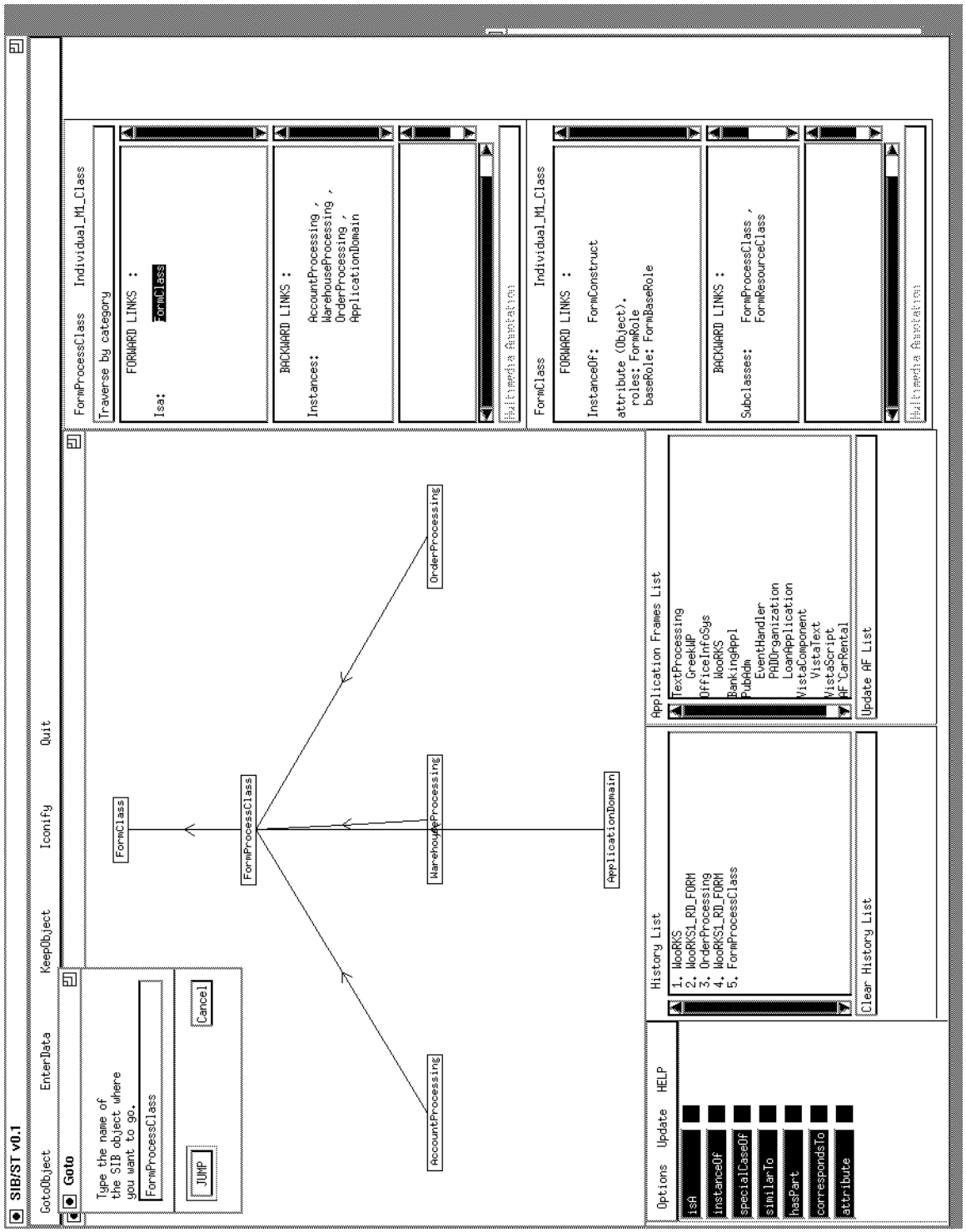


Figure 4.4: Visiting FormProcessClass and previewing its superclass, FormClass.

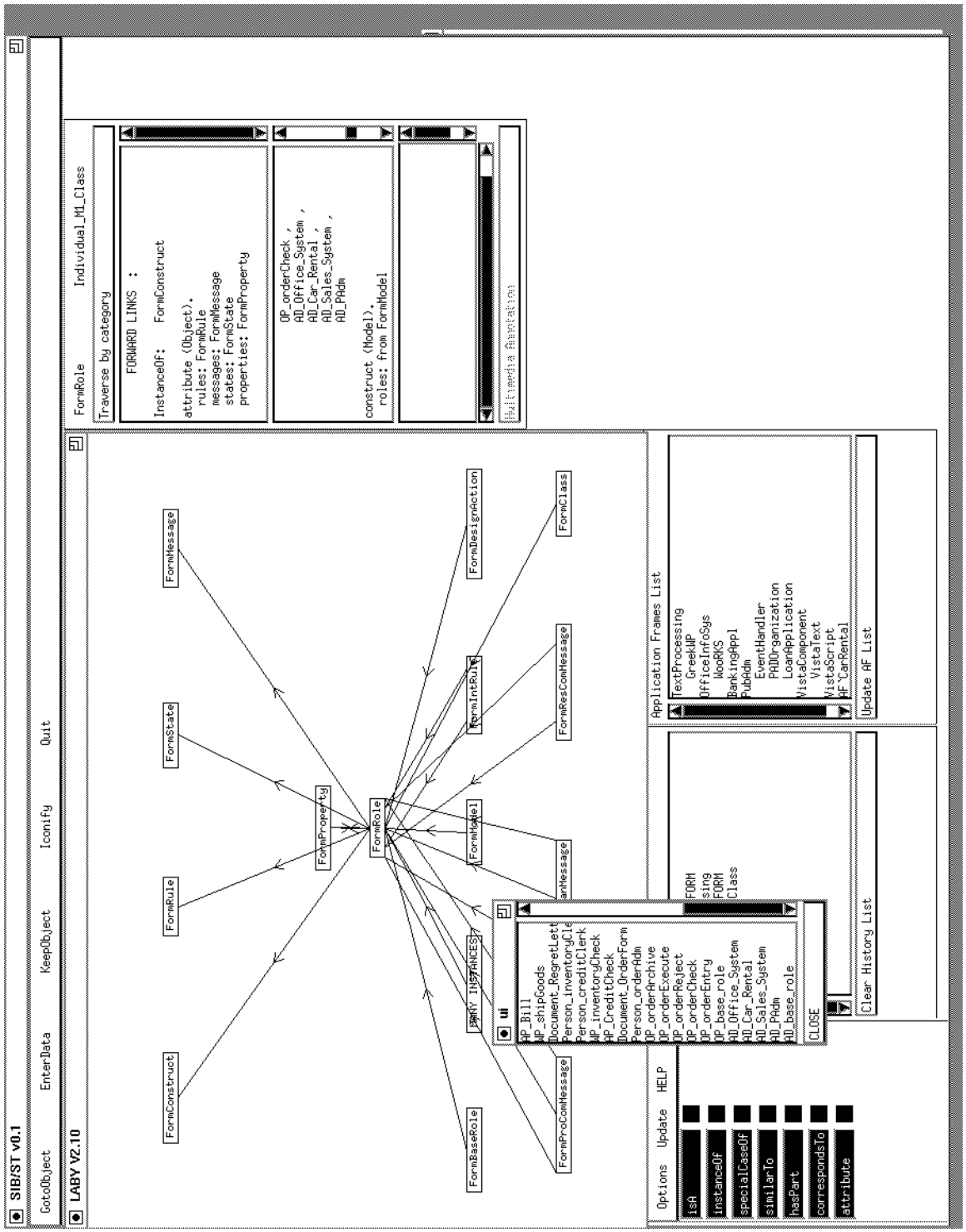


Figure 4.5: The Instance Box handles the instances of FormRole.

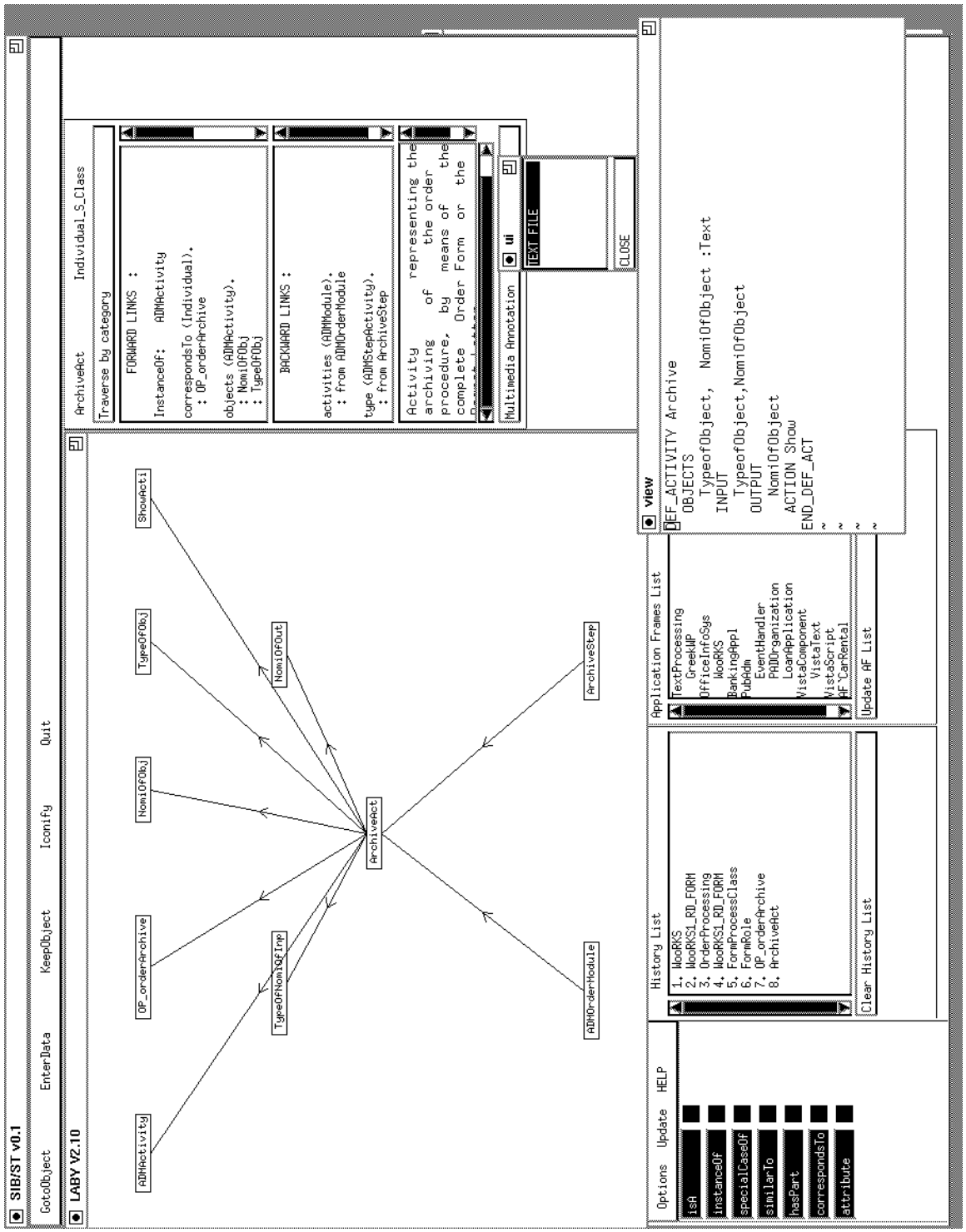


Figure 4.6: An annotation attached to ArchiveAct.

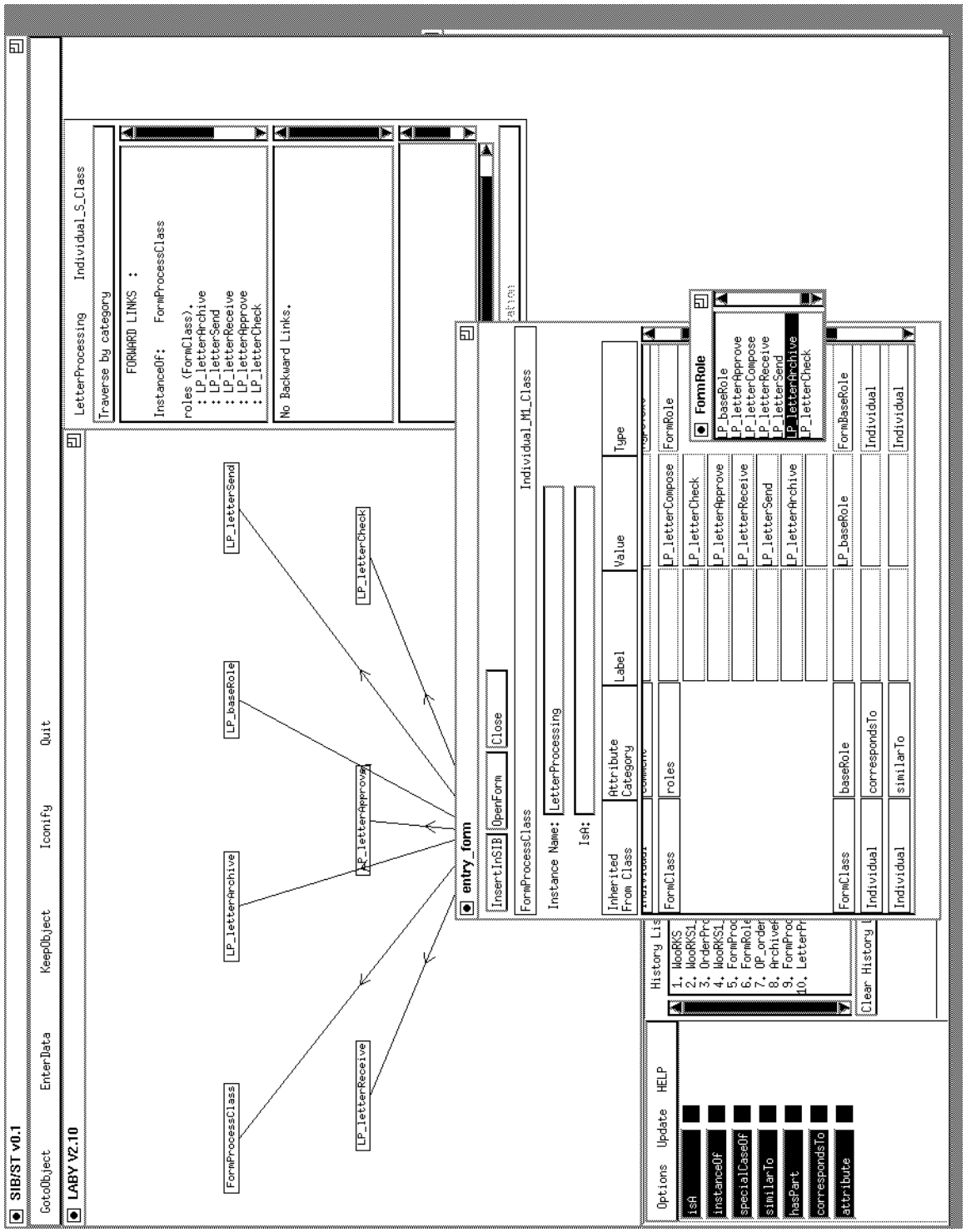


Figure 4.7: The EnterData facility used to insert LetterProcessing in the SIB.

# 5. Conclusions

The design of the Software Information Base integrates ideas and techniques from knowledge representation, graphics, databases, and software engineering to offer a basis for supporting software reuse. To make the integration possible, these techniques had to be enhanced and extended. Among the enhancements we note the efficient handling of large conceptual schemata in Telos, the rapid change of viewpoints and flexible presentation of large information bases supported by the constraint-based presentation mechanisms of LABY [Kate90], the tailor-made optimization of the SIB data management functions [Dado92], and the extension of the DAIDA framework for information systems representation [Jark92] by object and link type specifically dedicated to reuse. Scalability, portability, size-independent performance, and support of a multi-paradigm access with a carefully designed interface have all served as guiding principles of the prototype design and implementation.

The population of the SIB with software information about a real office application, a first test of the SIB concept, has demonstrated the breadth of reusability viewpoints handled effectively and the practical usefulness of the specific graphical support. Furthermore, performance results substantiate the technical choices made for the SIB. Ongoing experiments in different application domains and with different application development methodologies seem to indicate that this success is, in fact, generalizable.

However, the construction of application-specific SIBs requires more facilities than those offered by the current prototype. In particular, additional research is needed to address the problem of computer- assisted acquisition of similarity links. Ongoing work in this direction is reported in [Span92]. To guide the SIB developer (or to partially automate the job), reference models for different classes of applications would be of great value, analogously to the call for "shared ontologies" in DARPA's Knowledge Sharing Project [Pati92]. The extensibility of Telos is a big asset in the definition of such reference models. Moreover, besides defining the "right" classes of software objects and their relationships, there are problems with the description of individual objects to be classified. Firstly, we need to know how to describe them, secondly where to place them, and thirdly, how to do most of this automatically. A useful scheme of classification facets is presented in [Prie91]. Based on library science classification mechanisms, this scheme distinguishes facets such as the actions a system is to perform, its characteristics as an object that has been created and is stored, the main data structure it supports, or the usage for which it is intended. Based on this work and on semantic networks a classification model (the AIRS model) is proposed in [Oste92] that has been used for Ada and C libraries. The goal of this work is to provide for similarity-based retrieval and to automate the classification process. We have adapted this model to deal with the idiosyncrancies of the object-oriented nature of the languages used within the ITHACA setting.

Another potential, and desirable, extension of the SIB system could involve special-purpose tools for more "intelligent" support of the classification and the retrieval process. One such tool has been developed in prototype form ([KaVa92]). It employs case-based reasoning to adopt past experiences in searching for "analogous" software objects.

In the longer term, we want to experiment with the SIB model and system to also include other, more general, software-related information such as business plans, organizational strategies, and the like. Such external information provides non-functional requirements on the systems to be developed which can be exploited to steer the search process for reusable components. Non-functional requirements such as system performance, robustness, cost and security can also play an important role in selecting software components and in understanding the rationale behind software system structure, thereby facilitating their adaptation.

**Acknowledgements**

The Software Information Base system is the result of a large collective effort. We wish to acknowledge the deep influence of our colleagues within our respective research groups, particularly the SIB team, through long discussions and implementation work that had to be there but is often hard to appreciate. In particular, we acknowledge the invaluable contribution of the technical manager of the SIB team Dr. Martin Doerr, and that of Maria Theodoridou and Eleni Petra. We are grateful to Martin and also to Elena Pataki for help with the sections on implementation and the usage example respectively. Finally, we acknowledge the fruitful cooperation with members from other groups of ITHACA partners, namely, Siemens-Nixdorf (Germany), Universite de Geneve (Switzerland), TAO (Spain), National Technical University of Athens (Greece), Bull (France) and Politecnico di Milano (Italy).

# 6. References

[Ader90] Ader, M., Nierstrasz, O., McMahon, S., Mueller, G., Proefrock, A-K., "The Ithaca Technology: A Landscape for Object-Oriented Application Development", *Proceedings ESPRIT 1990 Conference*, Dordrecht NL,1990.

[Ader91] Ader, M., et al., Organization Model Reference Manual, ITHACA Report, ITHACA.Bull.91.D.1.4.#1.2, 1991.

[Atta81] Attardi, G. and Simi, M., "Completeness and Consistency of OMEGA, A Logic for Knowledge Representation", *Proceedings International Joint Conference on Artificial Intelligence*, Vancouver, 1981.

[Barl91] Barletta, R., "An Introduction to Case-Based Reasoning", *AI Expert*, August 1991.

[Bige88] Bigelow J., "Hypertext and CASE", *IEEE Software*, March 1988.

[Bigg87] Biggerstaff, T. et al., "Information Management Challenges in the Software Design Process", *IEEE Database Engineering,Vol 10*, March 1987.

[BiPe89] Biggerstaff T.J., and Perlis, A.J., **Software Reusability, Volume I: Concepts and Models, Volume 2: Applications and Experience**, ACM Press Frontier Series, Addison-Wesley, Reading, Massachusetts, 1989.

[BiRi89] Biggerstaff, T.J., and Richter, C., "Reusability Framework, Assessment, and Directions", in **Software Reusability: Volume I - Concepts and Models,** Biggerstaff, T.J., and Perlis, A.J., (eds), ACM Press Frontier Series, Addison-Wesley, Reading, Massachusetts, 1989, pp.1-17.

[Brod84] Brodie, M. and Ridjanovic, D., "On the Design and Specification of Database Transactions", in: Brodie, M., Mylopoulos, J. and Schmidt, J. (eds.), **On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages**, Springer-Verlag, 1984.

[Booc91] Booch, G., **Object Oriented Design with Applications**, Benjamin/Cummings Publishing Company, 1991.

[Borg89] Borgida, A., Brachman, R., McGuiness, D. and Resnick, L., "CLASSIC: A Structural Data Model for Objects" in Proceedings ACM SIGMOD Conference, Portland, 1989.

[Char92] Charalabidis, Y., Petra, E., and Vlodakis, G., "Populating Software Repositories: The SIB-WooRKS case", in *Proceedings of the ERCIM Workshop on Methods and Tools for Software Reuse*, October 1992.

[Chen76] Chen, P.P., "The Entity-Relationship Model: Towards a Unified View of Data", *ACM Transactions on Database Systems, Vol. 1, No. 1*, 1976.

[Conk87] Conklin J., "Hypertext: An Introduction and Survey", *IEEE Computer*, September 1987.

[CooL90] CooL 0.2 Language Description, ITHACA Report, ITHACA.SNI.90.L2.#4, 1990.

[Cons93] Constantopoulos, P., Doerr, M., and Vassiliou, Y., "Repositories for Software Reuse: The Software Information Base", to appear in *Proceedings of the IFIP WG 8.1 Conference on Information System Development Process*, September 1993.

[Dado92] Dadouris, C., Doerr, M, Kizlaridou, S. Mavroidis, D., Pataki, E., Theodorokis, E., Yeorgiannakis, G., Implementation of the SIB System, Report ITHACA.FORTH.92.E2.#2, Institute of Computer Science, FORTH, January 1992.

[Date83] Date, C.J., **An Introduction to Database Systems** (Vol II), Addison-Wesley, 1983.

[deAn91] De Antonellis, V., Pernici, B. and Samarati, P., "Object Orientation in the Analysis of Work Organization and Agent Cooperation", *Proceedings of the International Conference on Dynamic Aspects in Information Systems*, Washington, D.C., 1991.

[deMe91] de Mey, V. Junod, B., Renfer, S., Stadelmann, M., and Simitsek, I., "The Implementation of VISTA - a Visual Scripting Tool", in: Tsichritzis, D. (ed.), **Object Composition**, Centre Universitaire d'Informatique, Universite de Geneve, 1991, pp. 31-56.

[Deva91] Devanbu, P., R. J. Brachmann, P. Selfridge, B. Ballard, "LaSSIE: A Knowledge-Based Software Information System", *Communications of the ACM*, 34(5):34-49, May 1991.

[Ditt87] Dittrich, K. et al., "DAMOKLES - A Database System for Software Engineering Environments", *Lecture Notes in Computer Science*, Vol. 244, Springer, 1987.

[Espo92] Esposito, F., Malerba, D., Semeraro, G., "Classification in Noisy Environments Using a Distance Measure Between Structural Symbolic Descriptions", IEEE Transactions on Pattern Analysis and Machine Intelligence, 14(3), 1992.

[FiKe92] Fichman, R.G., and Kemerer, C.F., "Object-oriented and Conventional Analysis and Design Methodologies", in *IEEE Computer*, October 1992, pp.22-39.

[Fugi92] Fugini, M., O. Niestrasz, B. Pernici, "Application Development through Reuse: The ITHACA Tools Environment", *SIGOIS Bulletin, Vol.13, No.2*, August 1992.

[GaSc87] P. Garg, W. Scacchi, "On Designing Intelligent Hypertext Systems for Information Management in Software Engineering, DIF", *Proceedings of Hypertext 87*, pp. 409-431, November 1987.

[GaSc89] P. Garg, W. Scacchi, "ISHYS: Designing an Intelligent Software Hypertext System", *IEEE Expert*, pp.52-62, 1989.

[HuKi89] Hudson, S.E. and King, R., "Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System", *ACM Transactions on Database Systems 14(3)*, September 1989.

[Jark92] Jarke, M., Mylopoulos, J., Schmidt, J. and Vassiliou, Y., "DAIDA: An Environment for Evolving Information Systems", *ACM Transactions on Information Systems*, Vol. 10, No. 1, January 1992, pp. 1-50.

[Jark93] Jarke, M. (ed.), **Database Application Development with DAIDA**, Springer-Verlag, Heidelberg 1993.

[Jeus92] Jeusfeld, M., *Change Control in Deductive Object Bases*, DISKI Vol. 17, St. Augustin, Germany, INFIX Publ. (also Disseration University of Passau, in German), 1992.

[Jobe90] Jobes, B., "Repository comes of Age", *Database Programming and Design*, October 1990.

[Jone92] Jones, M.R., "Unveiling Repository Technology", *Database Programming and Design*, April 1992, pp.28-35.

[Kate90] Katevenis, M., Sorilos, T., Georgis, C. and Kalogerakis, P., *LABY User's Manual, Version 2.10*, ITHACA Report, ITHACA.FORTH.90.E.3.3.#7, 1990.

[Katz90] Katz, R., "Towards a Unified Framework for Version Modelling in Engineering Databases", *ACM Computing Surveys*, December 1990.

[KaVa92b] Katalagarianos, P., and Vassiliou, Y., "Employing Genericity and Case-Based Reasoning to Effectively Reuse Code", in *Proceedings of ICSC'92*, Hong Kong, December 1992.

[KhCo86] Khosafian S., Copeland G.P., "Object Identity", *Proceedings ACM OOPSLA '86.*

[KhFr88] Khosafian S., Frank D., "Implementation Techniques for Object Oriented Databases", *Proceedings AOODS*, Springer 1988.

[KoMc92] T. Korson, J. McGregor, "Technical Criteria for the Specification and Evaluation of Object-Oriented Libraries," *Software Engineering Journal*, March 1992.

[Krue92] Krueger, C.W., "Software Reuse", *ACM Computing Surveys, Vol.24, No.2*, 131-183, June 1992.

[Low88] Low C., "A Shared,Persistent Object Store", *Proceedings of the European Conference on Object-Oriented Programming,* 1988.

[MeLa87] Merrow T., Laursen J., "A Pragmatic System for Shared Persistent Objects", *Proceedings OOPSLA'87*, 1987.

[Meye85] Meyer, B., "Software Knowledge Bases", *Proceedings International Conference on Software Engineering*, London, 1985.

[Mich86] Michalski, R., "Learning from Observation: Conceptual Clustering", Machine Learning: An AI Approach, Vol. 1, Morgan Kaufmann Publ., 1986.

[Mylo90] Mylopoulos, J., Borgida, A., Jarke, M. and Koubarakis, M., "Telos: Representing Knowledge About Information Systems", *ACM Transaction on Information Systems,*, Vol. 8, No. 4, October 1990, pp. 325-362.

[Oste92] Ostertag, E., et al., "Computing Similarity in a Reuse Library System: An AI-Based Approach", *ACM Transactions on Software Engineering and Methodology, Vol.1, No.3*, July 1992, pp.205-228.

[Pati92] Patil, R., Fikes, R., Patel-Schneider, P., Mckay, D., Finin, T., Gruber, T. and Neches, R., "The DARPA Knowledge Sharing Effort: Progress Report", Proceedings Third International Conference on Knowledge Representation and Reasoning, Boston, November 1992.

[Pern90] Pernici, B., "Class Design and Metadesign", in Tsichritzis, D. (ed.), **Object Management,** Centre Universitaire d'Informatique, Universite de Geneve, 1990.

[Plot92] Plotkin, D., "Selecting a Repository", *Database Programming and Design*, April 1992, pp.28-35.

[Prie91] Prieto-Diaz, R., "Implementing Faceted Classification for Software Reuse", *Communications of the ACM*, May 1991.

[Proe92] Proefrock, K. et al, "Ithaca Final Report," December 1992.

[Rose91] Rose, T., Jarke, M., Gosek, M., Maltzahn, C., Nissen, H., "A Decision-Based Configuration Process Environment", Special Issue on Software Environments and Factories, *Software Engineering Journal 6, 5,* September 1991, pp. 332-346.

[Ross77] Ross, D. T., "Structured Analysis (SA): A Language for Communicating Ideas", *IEEE Transaction on Software Engineering*, January 1977.

[Russ88] Russel, S., "Analogy by Similarity" in Analogical Reasoning, Kluwer Academic Publ., 1988.

[Schw91] Schwanke, R., "An Intelligent Tool for Re-Engineering Software Modularity", *Proc.International Software Engineering Conference,* Austin, 1991.

[SkZd86] Skarra A.H., Zdonik S.B., "The Management of Changing Types in an Object-Oriented Database", *Proc. OOPSLA'86*, 1986.

[Span92] Spanoudakis, G., and Constantopoulos, P., "Similarity for Analogical Software Reuse: A Conceptual Modelling Approach", Proc. ERCIM Workshop on Methods and Tools for Software Reuse, Heraklion, October 1992.

[Tsic91] Tsichritzis, D. (ed.), **Object Composition,** Centre Universitaire d' Informatique, Universite de Geneve, 1991.

[Tver77] Tversky, A., "Features of Similarity", *Psychological Review,* July 1977.

[Wegn87] Wegner, P., "The Object-Oriented Classification Paradigm", in Research Directions in Object-Oriented Programming (Shriver, wegner, eds.), MIT Press, 1987.

[Wins87] Winston, M. and Chaffin, R., "A Taxonomy of Part-Whole Relations", *Cognitive Science 11*, 417-444, 1987.

[WKDa89] Won Kim, Kyung-Chang Kim, Dale A., "Indexing Techniques for Object Oriented Databases", in: **Object-Oriented Concepts,Databases, and Applications**, Kim, W. and Lochovsky, F. (eds.), ACM Press New York 1989