Chapter 7

Component Classification in the Software Information Base

Panos Constantopoulos and Martin Dörr

Abstract A key component in a reuse-oriented software development environment is an appropriate software repository. We present a repository system which supports the entire software development lifecycle, providing for the integrated and consistent representation, organization, storage, and management of reusable artefacts. The system can support multiple development and representation models and is dynamically adaptable to new ones. The chapter focuses on the facilities offered by the system for component classification, an important technique for retrieving reusable software. It is demonstrated that the inherently delicate and complex process of classification is streamlined and considerably facilitated by integrating it into a wider documentation environment and, especially, by connecting it with software static analysis. The benefits in terms of precision, consistency and ease of use can be significant for large scale applications.*

7.1 Introduction

Software reuse is a promising way of increasing productivity, assuring quality and meeting deadlines in software development. There are several, non-exclusive approaches to reuse, including organizational support, software libraries, object-oriented programming, AI-based methods for design reuse and process analysis.

* Work on the SIB was partly funded by the European Commission through ESPRIT project ITHACA. Partners in ITHACA were: Siemens-Nixdorf (Germany), University of Geneva (Switzerland), FORTH (Greece), Bull (France), TAO (Spain) and Datamont (Italy).

Panos Constantopoulos and Martin Dörr, "Component Classification in the Software Information Base," *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Eds.), pp. 177-200, Prentice Hall, 1995.

Reproduced with the permission of the Publisher, Prentice Hall (a Pearson Education company). This work is protected by copyright and may not be reproduced other than when downloaded and viewed on a single Central Processor Unit (CPU) for private use only. It is not otherwise to be reproduced or transmitted or made available on a network without prior written permission of Prentice Hall. All other rights reserved.

A common theme in all these approaches is that reuse concerns not only software code, but also design, requirements specifications and development processes. Supporting the communication of all these aspects of software development between the original developer and the reuser, and, furthermore, the cooperation within communities of software developers ("software communities" [14]), is a basic concern of reuse technology. Software repositories are key components in reuse-oriented software development environments [9] supporting the organization and management of software and of related information, as well as the selection and comprehension of relevant software and of development processes. In an orthogonal manner, object-oriented languages facilitate the development of reusable software components through encapsulation, data abstraction, instantiation, inheritance, genericity and strong typing. For broad, comprehensive surveys of reuse the reader is referred to [5] [18]. Krueger presents in [18] a taxonomy of reuse methods in terms of their ability to abstract, select, specialize (adapt) and integrate software artefacts.

In this chapter we assume that applications are developed using object-oriented technology, and that the software components of interest are mainly classes specified in an object-oriented programming language. As pointed out in [14], the management of large class collections introduces a number of problems concerning the representation of classes, in particular, the expression of structural and descriptive information, the representation of relationships and dependencies among classes in a collection, the selection and understanding of classes by appropriate querying and browsing facilities, and the support of class evolution.

Small to medium size collections of software classes can be organized by fairly simple schemes in the style of Smalltalk-80 [15]. Classes are hierarchically organized by inheritance and are grouped by functionality into possibly overlapping categories. The class browser allows the selection and exploration of reusable classes.

Various approaches have been proposed for addressing the selection problems arising in large collections. One such is the faceted classification scheme developed by Prieto-Diaz and Freeman [23]. In this scheme, components are classified according to six descriptors ("facets"), the values of which are hierarchically organized and on which a conceptual distance is defined. A variant of the faceted classification scheme, better suited for object-oriented software, was developed within the ESPRIT REBOOT project [17]. Other approaches to organizing software collections include: library cataloguing [16]; hypertext (DIF, [13]); object-oriented libraries (Eiffel [20], Objective-C); ER and extended models (IBM Repository [19], Lassie of AT&T [11]); and hybrid approaches (e.g. SIB [8]).

The Software Information Base (SIB) is a repository system, developed within the ESPRIT ITHACA project, that stores information about the entire software lifecycle. The SIB offers a uniform representation scheme for the various artefacts and concepts involved in the different stages of the software lifecycle; the scheme can be extended to accommodate new ones. It also supports multiple forms of presentation, depending on the tool using the particular artefact. Finally, it provides querying, browsing and filtering mechanisms for selecting and understanding artefacts, and interfaces to other software development tools.

In this chapter we first give an overview of the SIB and of its concepts in section 7.2. We examine the querying and browsing capabilities of the SIB in section 7.3. The SIB's classification scheme is described in section 7.4; section 7.5 explains how the classification of software artefacts is automated, whereas section 7.6 reports on our experiences with the SIB. We conclude with perspectives for future work.

7.2 The Software Information Base

7.2.1 General Concepts

The SIB is structured as an attributed directed graph, with nodes and links respectively representing descriptions of software artefacts (objects) and relations between them. There are three kinds of descriptions, namely:

1. requirements descriptions (RD);
2. design descriptions (DD); and
3. implementation descriptions (ID).

These descriptions provide three corresponding views of a software object:

1. an application view, according to a requirements specification model (e.g. SADT);
2. a system view, according to a design specification model (e.g. DFD); and
3. an implementation view, according to an implementation model (e.g. set of C++ classes along with documentation).

Descriptions can be simple or composite, consisting of other descriptions. The term *descriptions* reflects the fact that these entities only describe software objects. The objects themselves reside outside the SIB (e.g. in a Unix file storing a C++ program), accessible from the corresponding descriptions.

There are several kinds of relationship between descriptions or parts of descriptions serving a variety of purposes:

- *Flexibility* in *defining* or *modifying* types of artefacts and relationships, or even description models, accomplished through multiple instantiation and a series of instantiation levels.
- Classification of artefacts and relationships in *generalization/specialization* hierarchies supporting multiple strict inheritance.
- expression of *semantic* and *structural* relationships between artefacts, including aggregation, correspondence, genericity and similarity.
- expression of *user-defined* and *informal links* — including links for hypertext navigation, annotations, and for defining version derivation graphs.
- *grouping* of software artefacts descriptions into larger functional units.

An important concept in the SIB is the *application frame (AF)*. Application frames represent complete systems or families of systems and comprise (*hasPart*) at least one implementation and optional design and requirements descriptions. AFs are further distinguished into specific and generic (SAFs and GAFs) while the RDs, DDs and IDs of an AF should be considered as groupings of such descriptions (i.e. other associations).

A SAF describes a complete system (be it a linear programming package, a text processor or an airline reservation system) and includes exactly one ID. A GAF is an abstraction of a collection of systems pertinent to a particular application and includes one RD, one or more DDs and one or more IDs for each DD. Application frames play a key role in the reuse-oriented software development lifecycle envisaged in ITHACA. Generic components and applications are produced by *application engineers*. These are represented by GAFs and constitute a core of, presumably good quality, components and applications which are configured and adapted to fit particular needs by *application developers*. Such derived, specific systems are represented by SAFs. For more on the ITHACA application development methodology and the role of application frames see [10] [9].

The representation language employed in the SIB is Telos [21]: a conceptual modelling language in the family of entity–relationship models [7]. The main reason for choosing Telos over other E-R extensions, such as those used by the PCTE+ OMS or the IBM Repository Manager, MVS, is that it supports unlimited instantiation levels and treats attributes as objects in their own right (which, therefore, can also have attributes). These features account for great expressiveness and easy schema extension, and are fully exploited in the SIB.

7.2.2 Relationships Between Software Artefacts

Relationships are essential for the classification and retrieval of software artefacts. We therefore elaborate on each kind of link and indicate, when appropriate, how they support the querying and browsing activities in the SIB.

Attribution

Attribution is represented by *attribute* links. This is a general, rather unconstrained representation of semantic relations, whereby the attributes of a description are defined to be instances of other descriptions. An attribute can have zero or more values. Consider the following example:

```
Description SoftwareObject with
    attributes
        author : Person
        version : VersionNumber
```

SoftwareObject has attributes author and version whose values are instances of Person and VersionNumber respectively. Dynamic properties, such as ‘calls’ relations of methods and procedures, also fall into this category.

Aggregation

Aggregation is represented by *hasPart* links. This relates an object to its components. For example:

```
Description SoftwareObject with
          ...
           hasPart
                components: SoftwareObject
```

The components of an object have a distinct role in the function of the object and any possible changes to them affect the aggregate object as well (e.g. new version).

Classification

Classification (converse *instantiation)* is represented by *instanceOf* links. Objects sharing common properties can be grouped into classes. An object can belong to more than one class. Classes themselves are treated as generic objects, which, in turn, will be instances of other, more generic objects (so-called "meta-classes"). In fact, every SIB object has to be declared as an instance of at least one class. Effectively, an infinite classification hierarchy is established starting with objects that have no instances of their own, called tokens. Instantiation of a class involves instantiating all the associated semantic relations. Thus relations are treated as objects themselves. For example:

```
Description BankIS instanceOf SoftwareObject with
          author : Panos
          version : 0.1
  components : CustomerAccounts, Credit, Investments
```

The attribute and hasPart links of BankIS are instances of the corresponding attribute and components links of SoftwareObject.

Classification is perhaps the most important modelling mechanism in the SIB. [33] gives a detailed account of the construction of models and descriptions in the SIB.

Generalization

Generalization (converse *specialization*) is represented by *isA* links. This allows multiple, strict inheritance of properties between classes leading to the creation of multiple generalization hierarchies. A class inherits all the attributes of its superclasses (possibly more than one — multiple inheritance); however, inherited properties can only be constrained, not overridden (strict inheritance).

Correspondence

Correspondence is represented by *correspondsTo* links. A software object can have zero or more associated requirements, design and implementation descriptions. Correspondence relations concern the identity of an object described by different descriptions and can have as parts other correspondence relations between parts of the corresponding descriptions. Correspondence links actually indicate that the descriptions they link together describe the same object from different perspectives. The correspondences of the parts need not be one-to-one. For instance, a requirements specification may correspond to

more than one design and a design may have more than one alternative implementation. Similarly, a single implementation could correspond to more than one design entity. Application Frames are an important type of controlled correspondence in the SIB.

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 [32] and AI, most relevantly to this work in case-based reasoning [4]. Within the context of object-oriented systems, similarity has been viewed as a form of generalization [34]. Alternatively, it has been interpreted as degree of affinity with respect to various relations, providing the foundation for the dynamically changing presentation of related objects within a browser (see chapter 9). Its applications include the support of *approximate* retrieval with respect to a software repository as well as the re-engineering of software systems [27].

We are primarily interested in similarity links that can be computed automatically from information that is loaded into the SIB. For added flexibility, however, user-defined similarity links are also supported. 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. For details on the similarity analysis of SIB descriptions see [29].

Genericity

Genericity is represented by *specialCaseOf* links. This relation is defined only between application frames to denote that one application frame is less parameterized than another. For example, a bank accounting and a hotel accounting application frame could both be derived from a more general, parametric accounting application frame.

Informal and user-defined links

When users have foreseeable needs for other types of links they can define them using the attribute definition facility of Telos. For instance, versioning can be modelled by special correspondence links labelled *derivedFrom*. Furthermore, random needs for representation and reference can be served by informal links, such as hypertext links which allow the attachment of multimedia annotations to SIB objects.

Association

Association is an encapsulation mechanism intended to allow the grouping of descriptions that together play a functional role [6]. It associates a set of descriptions with a particular symbol table:

Association = (setOfDescriptions, symbolTable)

The contents of an association can only be accessed through the entry points supplied in its symbol table. 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. The SIB itself is a global association containing all objects included in any association. Its symbol table contains all the external names of every object. Name conflicts can be resolved by a precedence rule.

Associations can be derived from other associations through queries or set operations. Furthermore, associations can be considered as materialized views. Non-materialized views, or simply *views*, differ from associations in that they cannot be updated directly, but rather, through updates of the associations which they are derived from.

7.3 Information Retrieval and User Interface

7.3.1 Querying and Browsing

The selection of software descriptions from the SIB is accomplished through the *selection tool* (ST) in terms of an iterative process consisting of retrieval and browsing steps. 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.

The basic selection functions of the SIB are:

Retrieve: Queries $\times$ Associations $\rightarrow$ P(Descriptions $\times$ Weights)
Browse: Identifiers $\times$ P(Links $\times$ Depths) $\times$ Associations $\rightarrow$ Views

The *Retrieve* function takes as input a (compound, in general non-Boolean) query and an association, and returns a subset of the associated descriptions with weights attached, indicating the degree to which each description in the answer set matches the query. Non-Boolean queries are based on similarity. Queries are formulated in terms of the query primitives offered by the Programmatic Query Interface. A set of queries of particular significance can be preformulated and offered as menu options, thus providing maximum ease-of-use and efficiency for frequent retrieval operations.

Browsing begins with a particular SIB description which is the current focus of attention (called the *current object*) and produces a view of a *neighbourhood* of the current ob-

ject within a given association. Since the SIB has a network structure, the neighbourhood of the current object is defined in terms of incoming and outgoing links of interest. Moreover, the size of the neighbourhood 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 centred around the current object.

When the depth control parameters are all equal to 1, a *star view* results, showing the current object at the centre surrounded by objects directly connected to it through links of the selected types. This is the simplest and smallest neighbourhood of an object, in topological terms, with a controllable population. 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, *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 the current object with respect to one or more link types is displayed. Such a browse operation can display, for example, the call graph (forward or backward) of a given routine.

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, "canned" queries. Browsing commands and explicit queries can also be issued through appropriate interfaces from external tools.

7.3.2 Implementation

An application-scale SIB system has been implemented at the Institute of Computer Science, FORTH, and is available to other sites for experimentation* (see figure 7.1).

The user interface supports menu-guided and forms-based query formulation with graphical and textual presentation of the answer sets, as well as graphical browsing in a hypertext-like manner. A hypertext annotation mechanism is also provided. Menu titles, menu layout and domain-specific queries are user-configurable.

A forms-based interactive data entry facility is available for entering data and schema information in a uniform manner. This facility automatically adapts itself to the structure of the various classes and subclasses byemploying the schema information. Furthermore, it is customizable to application-specific tasks, such as classification of items, addition of descriptive elements, etc.

* For details, consult the WWW page for this book (see the preface).

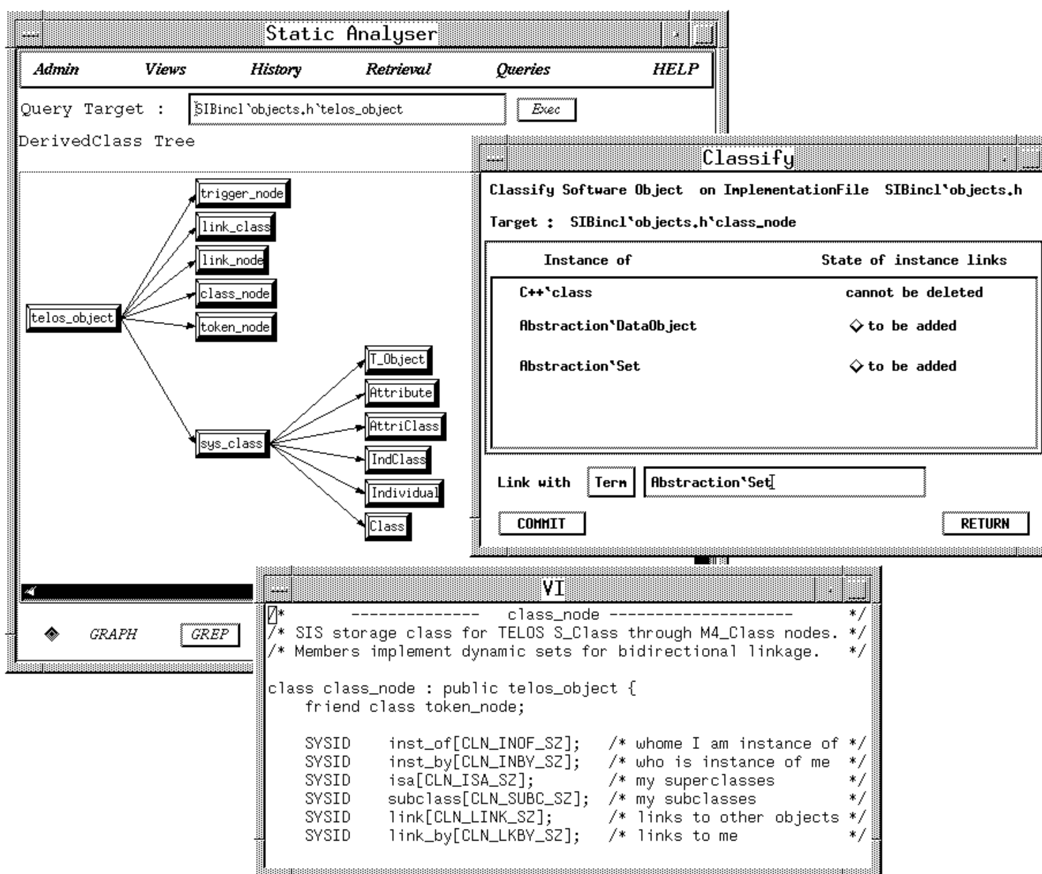

Figure 7.1 *SIB static analyzer and class management system.*

Any item in the SIB may reference a multimedia object, comprising images, video, sound or text, stored externally. The SIB recognizes such references and automatically generates calls to the appropriate presentation tools with the respective parameters, which results in a synchronous display of the multimedia object.

The SIB is optimized for referential access and large amounts of highly structured data, especially for network structures consisting of a large variety of classes, rather than relatively few classes with large populations per class (typical in a DBMS). Recursive queries on aggregational hierarchies, classification hierarchies, and retrieval of graph structures, such as flow-charts or state-transition diagrams, play a more important role than access by value conditions. A transitive closure with cycle detection of a binary tree with 1024 nodes can be retrieved in 2 seconds on a Sun SPARC Station. The performance of the SIB in look-up and traversal exceeds that of modern relational systems by one and two orders of magnitude respectively. This allows for real-time queries that would be prohibitive with traditional databases.

Data entry speed is acceptable: 10,000 references are loaded in batch mode in 3 minutes, and 500,000 in 2.5 hours on a SPARC. Both examples were measured with real

application data from static analysis of a medium (30.000 code lines) and a very large application (2.5 million code lines). The theoretical capacity limit is 1 billion references. The design of the internal catalogue structures is fully scalable.

For more on the SIB, the interested reader is referred to [8] [9].

7.4 The Classification Scheme

7.4.1 Principles

Given a set of entities (objects, concepts) represented by descriptors (keywords), the grouping of those entities into disjoint classes according to some criterion of descriptor matching is called *classification*. Matching may express some kind of semantic similarity. A *classification scheme* determines how to perform classification in a given setting, prescribing the sets of descriptors and possible internal ordering, matching criteria, and rules for class assignment.

Depending on the number of descriptors used, a classification scheme can be uni- or multi-dimensional. An example of a unidimensional scheme is the Universal Decimal Classification (see [26]). In library science, multidimensional (*faceted*) classification, was introduced by Ranghanathan [25], breaking down information into a number of categories thus addressing corresponding aspects of the classified entities. These aspects are called *facets*.

Prieto-Diaz and Freeman developed a faceted classification scheme for software reuse [23] [24] in which they use six facets to describe software: *function*, *object*, *medium/agent*, *system type*, *functional area*, and *setting*. They mainly describe component functionality, the last three facets pertaining to the internal and external environment. Each facet has a *term space*, i.e. a fixed set of legal values (*concepts*), in the sense of a controlled vocabulary, and an extensible set of *user terms*. Concepts are organized by a directed acyclic specialization relation, and terms are assigned as leaves to concepts. Subjective conceptual distances between concepts and terms are defined, to support retrieving software components by their degree of matching.

A variant of the scheme of Prieto-Diaz and Freeman was developed in the ESPRIT REBOOT project [17] [28] [22] [31]. This scheme comprises four facets, better suited for describing object-oriented components: *abstraction*, *operation*, *operates-on* and *dependency.* The first three are analogous to subject, verb and object in a natural language sentence describing component functionality, while the fourth is the counterpart of the three environmental facets of the Prieto-Diaz and Freeman scheme. The term spaces are also structured by relations such as specialization and synonymy. A conceptual distance between terms is defined, which, like that of Prieto-Diaz and Freeman, is the outcome of human assessment. Neither Prieto-Diaz and Freeman nor REBOOT relate the derivation of classification terms to the knowledge of structural dependencies between software components. In [28], however, such a connection is suggested as potentially useful.

In the SIB classification scheme the REBOOT facets are adopted, except that facets are assigned not necessarily to a class as a whole but, rather, to the relevant parts. Specifically, the contents of the SIB classification facets are as follows:

Abstraction

Abstraction terms are nouns representing *active* object types. Typically these abstractions indicate the role that the object plays in its interactions with other objects of an application. An object-oriented software class as a whole is assigned an abstraction, such as ‘String’, ‘Set’, ‘WindowSystem’, ‘Index’ or ‘NameList’. Abstraction terms do not include expressions that denote processing, such as ‘String concatenation’ or ‘String conversion’. Since object types are assumed to be active, the Abstraction terms do not reflect processing in general either (e.g. ‘String manipulation’).

Operation

Operation terms are verbal types representing specific activities. The active part of a class comprises its methods. Hence we associate Operation terms with each individual method responsible for an activity, e.g. ‘Solve’, ‘Invert’, ‘Lock-Unlock’, ‘Open-Close’. Pairs of inverse properties, such as ‘Open-Close’, are regarded as one term, to keep the term space small.

Operates-On

Besides operating on the class to which it belongs, a method operates on its parameters. In object-oriented design, non-trivial parameters belong to classes. (Methods may also directly access input/output devices, which may or may not be represented as objects.) Operates-On terms are nouns representing the object types acted on by methods, including Abstractions, basic data types and devices. Note that Operates-On is a superset of Abstraction and that the abstraction of a class must be a default ‘Operates-On’ for its own operations. Operates-On represents the role an object type plays with respect to other types.

Dependency

Dependency terms represent environmental conditions, such as hardware, operating system or language. It is good practice in software development groups to test and release complete libraries for a certain environment. Accordingly we assign Dependency terms to class libraries as a whole. The classes of the library are then indirectly linked to a dependency through the library itself. Each combination of programming language, system software and hardware forms a different environment. Dependency terms are provided in the SIB which reflect single environmental conditions, as well as combinations of those. For instance, a library tested, for example, on (SINIX ODT1.1, AT&T C++ 3.0B, SNI WX200), and (SINIX 5.41, CooL 2.1, SNI WX200) does not necessarily run on (SINIX 5.41, AT&T C++ 3.0B, SNI WX200). Such triples are terms by themselves in the SIB, the constituents of which represent their immediate higher terms. Thus retrieval is possible by the triple itself, as well as by simple terms, e.g. SINIX 5.41, Unix, C++, etc. (see figure 7.2).

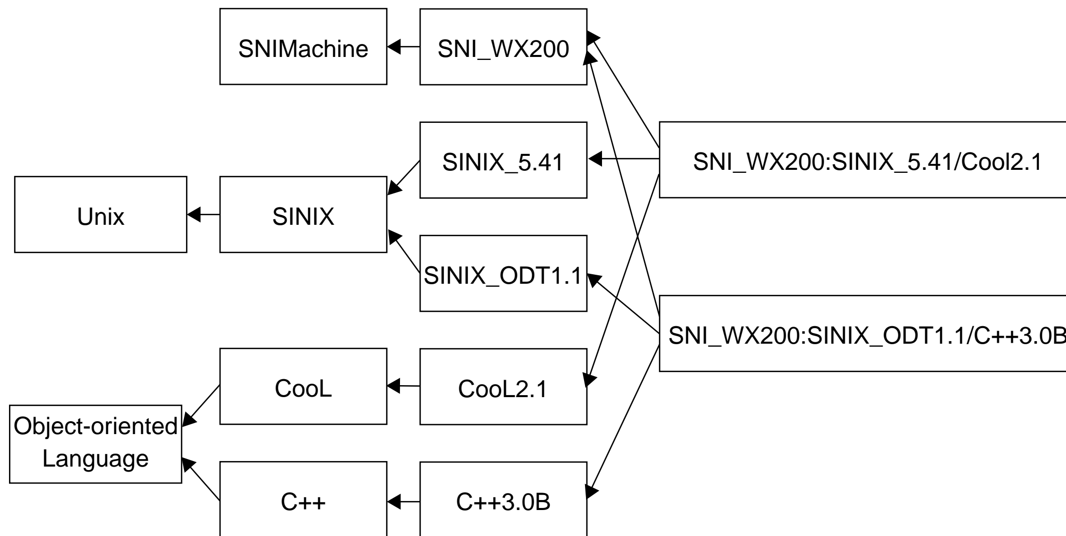

Figure 7.2 *The isA hierarchy of combinatory Dependency terms.*

7.4.2 Classification Hierarchies in the SIB

Facets are represented as meta-classes in the SIB. The terms, i.e. the values of a facet, are instances of that facet, and are therefore simple classes. The instances of those classes are the software objects sharing the functional property the term denotes. The assignment of a term to a software object is accomplished by declaring the object to be an instance of the term.

Note that facet terms reflect the *functional* role of components as they cooperate in a process, as distinguished from structural relations or user application tasks. For instance, 'C++ class', 'Menu item', 'Selection of goods, clients, or accounts' may be respectively the structural, functional and application roles of one software object. These other two roles are also very pertinent for reuse. In the SIB we take advantage of such information both independently and jointly with functional classification, as we shall see below. On the other hand, some essential functional parts of a software object are and should be hidden from the user, hence they do not have any application task associated with them.

In addition to a functional classification scheme, like the one discussed here, one can independently develop a classification scheme with respect to structural aspects of the programming language, or other criteria. Concurrent classification of software objects according to more than one scheme is supported by the SIB. Technically, the assignment of terms from several schemes is performed by multiple classification: a component is declared to be an instance of all the relevant terms.

The term space for each facet is partially ordered by a specialization/generalization relation (isA) which, in the SIB, obeys multiple strict inheritance. This organization has a

number of advantages. It minimizes the data entry involved in describing a software component, since each term inherits all its predecessors. In addition, the probability of inconsistent classification is limited. Multiple isA relations express multiple independent properties. By contrast, simple isA relations generate pure hierarchical structures which are not flexible enough for expressing general, multifaceted relationships. If we interpret the term space as a set of classes semantically ordered by isA relations derived from their implicit properties, assuming strict inheritance of those properties, the terms lose their linguistic nature and become concepts. *Homonyms*, i.e. instances of the same word with different meanings, must then be assigned different terms. For example, spectral 'radius' and circle 'radius', law and 'order' and warehouse 'order' do not share properties. The recall of such a concept-based system is superior to a linguistic one (see [30]).

No distinction in nature is made between leaf terms and higher terms. The granularity of analysis depends very much on the breadth of the domains classified, and should be dynamically adaptable to the contents of the repository. As is generally accepted in the literature, term spaces are kept small in order to help the user become quickly acquainted with the applicable terms for a given problem. Retrieved components can subsequently be reviewed efficiently by browsing.

In order to combine discipline with linguistic flexibility in developing term spaces, synonyms are introduced. Two different words are *synonyms* if they have the same meaning, such as 'alter' and 'change' in the Unix system call manual. Preferred words are selected as terms for inclusion in the isA-structured term space. Synonyms are attached to those through the attribute category *synonym* specifically defined in the SIB. Thus access is possible by all synonymous words and term, while vocabulary control is maintained.

Multiple inheritance expresses multiplicity of nature of a term itself. For instance, 'Copy' has the properties of both 'Put' and 'Get'. We adopt the principle that for any group of terms sharing some implicit property, there should be a higher term with this property: 'Get', 'Put', 'I/O' share a property 'transfer', whereas 'Put', 'Update' share 'Modify', etc. Arbitrary decisions on term placement in a hierarchy can thus be avoided. On the other hand, as any conceptual distance or similarity is based on some sharing of properties, those notions become closely related to the higher terms structure.

Multiplicity of nature at the item level (components), not resulting from intrinsic properties of the terms, is expressed through multiple instantiation (assignment of terms), e.g. a method doing 'Lock' and 'Update'. It turns out that the benefits from the specialization (isA) structure of term spaces are fully obtained if items are only assigned to leaf terms. Nevertheless, the system is robust to a dynamic refinement of the term space, whereby leaf terms may become higher terms. Items assigned to higher terms can be treated by the retrieval query as possible candidates for all leaf terms under it, with a decreasing priority according to the number of levels between them.

The isA organization facilitates exploring, understanding and retrieving terms. Naturally, alphabetical browsing and retrieval by lexical pattern matching are also provided. Finally, a "conceptual distance" (conversely, similarity) can be defined as a suitable metric over the term space partially ordered by the isA relation [29]. The advantage of such a metric is that its computation requires no user input, as it effectively relies on the intrinsic

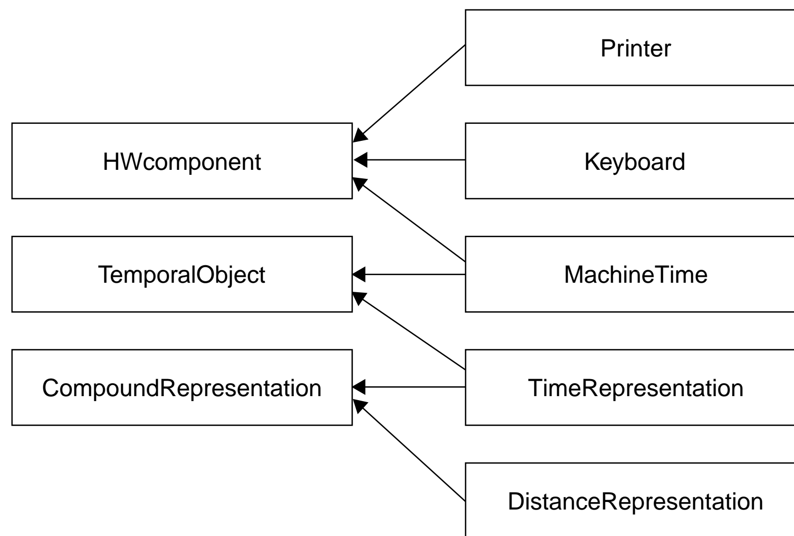

Figure 7.3 *The isA hierarchy environment of Abstraction 'TimeRepresentation'.*

properties of the common higher terms. To which degree this notion of measure can be exploited to improve or normalize the hierarchy itself is a topic of further research.

7.4.3 Example

Let us consider the abstractions of a class 'Time', which handles arithmetic with years, hours, minutes, etc. On the one hand, it has to do with the representation of time values; on the other hand, it does not relate to actual time. We therefore choose the term 'TimeRepresentation'. This term has two higher terms: 'TemporalObject', and 'CompoundRepresentation'. By 'CompoundRepresentation' we denote systems of measurement using different units for different orders of magnitude, such as miles, yards, etc. Another specialization of 'TemporalObject' is 'MachineTime'.

A 'TimeRepresentation' class may be directly used, or in conjunction with a 'MachineTime' class to measure elapsed time. This conforms to the initial intention of such a class. Note that we could easily change unit names and conversion factors between units to adapt such a class to handle miles, yards, etc. In this case we reuse the algorithm or structure of a specific solution. This property is intrinsic to a time representation module, and we express it by the higher term 'CompoundRepresentation'.

This example demonstrates how multiple inheritance can serve to bring related terms together, and how a careful analysis of implicit properties of terms may help to support reuse in ways the developer did not originally have in mind (see figure 7.3). Since the development of generic modules is regarded to be desirable for reuse, any support for detecting candidates to be generalized or parameterized is valuable.

7.5 Streamlining the Classification Process

7.5.1 Static Class Analysis

The SIB stores various kinds of structural and descriptive information about software components. In particular, at the implementation description level, it stores the results of static analysis performed by program parsers. Given a programming language, a corresponding implementation description model defines a set of entities, such as class, method, parameter, source file, and relations between these entities, such as defines, calls, user-of. The static analysis data of a given component are entered as instances of the model concepts. (This information is useful enough in its own right that a version of the SIB has been developed purely as static analyzer.) Static analysis information is also useful for streamlining the classification process.

We distinguish classification into *direct*, which is assigned explicitly to an entity, and *derived*, which is defined by means of queries. Minimizing direct classification not only saves human effort, but also improves consistency when software or term space modifications take place. Static analysis allows for an automatic mapping of information (methods, parameters, etc.) to classification facets and terms.

Abstractions are associated to classes, and Operations are associated to methods. Procedures and operators are treated like methods, if they are connected to a class via friend declarations. Otherwise, an additional class, such as 'Procedure_group' is introduced for their classification. The operations of a class are derived by queries through the links indicating the methods belonging to that class. The explicit correspondence of methods and Operations facilitates maintenance and consistency of code and classification terms. The Operates-On terms of a class are also derived, and include the abstraction of the class itself (since its methods can access its instances) and the parameter types of its methods, be they abstractions (i.e. other classes) or basic data types. (The assignment of terms representing devices, system calls, etc., to methods is done manually at present.) In a linguistic sense, Operates-On is the direct object of the Operation verb. Operates-On is at first hand a property of the method, or even more precisely of the instantiation of a specific operation in a method, and only in a wider sense a property of the class as a whole.

Dependency terms are assigned to libraries and applications. Therefore, the dependencies of a class are derived by relating library files with the classes they contain.

7.5.2 Derived Classification

A number of derivation paths are used, in either direction, depending on whether the objective is to find a class or the valid terms for a class. An example comprising all those paths is given in the next section. The following is the complete list of relevant path elements:

- From synonyms to established terms through the 'synonym of' link.

- From class terms to higher terms through the 'isA' link.
- From Abstraction terms of classes to methods as 'Operates-On' through the inverse 'has parameter' link.
- From method terms to classes through the inverse 'has method' link.
- From class terms to derived classes, in the sense of the PL, through the inverse of the 'has parent' or 'has supertype' link.
- From Dependency terms to classes through the 'Library.runs_on' – 'Library.has_-file' – 'class.defined_in' path.

Note that the direct assignment of terms is done not to software classes but to more finely-grained entities (e.g. methods) that are structurally related to them. In this way it is sufficient, in most practical cases, to assign one term to each entity. Furthermore, static analysis information can support the automatic extraction of classification terms from formalized source code comments.

When creating term spaces it is important to maintain semantic links between Operation terms and Abstraction terms, in particular:

1. which legal operations belong to an abstraction; and
2. which application domain an operation term is intended for.

The first kind of constraint should naturally be represented by linking abstractions with their legal operations. Creating higher operation terms would be unnecessary. For example, Operation'truncate is an operation applicable to both strings and files. This should be indicated by links from Abstraction'file and Abstraction'string to Operation'truncate. Introducing, say, Operation'string_operations and Operation'file_operations as higher terms, to which Operation'truncate would be isA related, conveys no information on the *nature* of Operation'truncate.

The second kind of constraint introduces a problem related to homonyms. These are handled by adding prefixes to the terms, so that the homonyms effectively obtain unique names in the SIB. Besides, they preserve the homonym character in the last part of the word, which allows access by substring matching. However, great care should be taken not to create substructures in the term space on the basis of homonyms (more precisely: the homonymous parts of terms), which may prove semantically wrong. For example, a substructure including Abstraction'order along with its specializations Abstraction'warehouse_order and Abstraction'serial_order is not based on common semantics as expected. By contrast, Abstraction'warehouse_order isA Abstraction'commerce and Abstraction'serial_order isA Abstraction'memory_management are semantically correct.

7.6 Experiences

7.6.1 The Classification Process

Classification is an iterative process. The user understands the functionality of a part of a component by studying (through the browser) documentation, static analysis data, and/or

the code itself, supported in each step by the SIB, matching it with terms in the term space. Term understanding is supported by the linguistic form of the term, its position in the hierarchy, text comments on its meaning, or use in the classification of similar code known to the user. The user must decide if a given term matches with the component and if the term is specific enough. If not, a new term must be introduced in agreement with a group of developers responsible for the term space maintenance. With use and experience, the upper parts of the term space become increasingly stable and complete.

A user should be aware not only of the meaning of terms, but also of their *quality* (i.e. for retrieval purposes), which leads to the need to know the principles under which terms are created. The following general criteria are proposed for selecting terms in a given domain of interest [12] [22]:

- Terms should be *well-known words*, usually technical terms or expressions, widely accepted in the software engineering community, or at least by experts in the particular domain of interest (object-oriented development).
- Terms should have *clear meanings*, relative and easily associated to the concepts conveyed by their specializations or generalizations, in the classification structure. Moreover, they should be *distinct* and precise, in order to facilitate the direct linking of the component to the corresponding classification term.
- Terms should also be *general* enough, in the sense that a term may encompass more than one specialized term in the classification structure. In other words, every term should be used to address more than one component, or a specific (under some semantic criteria), set of components. Keeping a set of terms general — therefore small enough, and expressive at the same time, thus useful for the reuse process — is one of the basic and most difficult tasks in classification. Conversely, keeping a large term space usually means confusion for suppliers and reusers of components, inconvenient browsing, poor search performance, etc.
- Redundancy should be avoided, in the sense that there should be no two terms with very close meaning in the same classification hierarchy. If this happens, then they should be related only with synonym relationship, with the most representative term present in the classification hierarchy.

As these criteria are generally conflicting, the implementation of an effective term space requires striking a judicious balance among them: a non-trivial task.

7.6.2 An Example

We draw an example from the classification developed for the Colibri class library [3] of the CooL language environment. CooL [2] is an object-oriented programming language developed at Siemens-Nixdorf within the ESPRIT ITHACA project. We demonstrate the selection and the assignment of terms, and the resulting valid terms by derivation.

Consider the following partial listing of the classes ‘Date’ and ‘DateRepr’:

```
--                              -*- Mode: Cool -*-
-- Date.t --
--
-- PURPOSE
      |     Date is an object type representing a calendar entry        |
      |     consisting of year, month, and day. This object type        |
      |     offers methods to construct, modify and actualize an        |
      |     object and to get information about an object. Further      |
      |     methods deal with arithmetic operations and                 |
      |     predicates                                                  |
-- TABLE OF CONTENTS
REFER Duration, Interval, time;
TYPE Date = OBJECT (    IN Year : INT,
                        IN Month : INT,
                        IN Day : INT)
-- ---------------------------------------------------------------------------
-- 2. Actual Date
-- ---------------------------------------------------------------------------
METHOD SetToActualDate;
      -- ---------------------------------------------------------------------
      -- Set this date to the actual date.
      -- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------------
-- 4. Selective access
-- ---------------------------------------------------------------------------
METHOD GetYear : INT;
METHOD GetMonth : INT;
METHOD GetDay : INT;
      -- ---------------------------------------------------------------------
      -- Return the specific information of this date
      -- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------------
-- 5. Arithmetic operations
-- ---------------------------------------------------------------------------
METHOD Add (IN Extent : Duration);
METHOD Subtract (IN Extent : Duration);
-- ---------------------------------------------------------------------------
 -- Add or subtract an extent from this date.
-- ---------------------------------------------------------------------------

END OBJECT;
-- ---------------------------------------------------------------------------

--                              -*- Mode: Cool -*-
-- DateRepr.t --
--
-- PURPOSE
      |     DateRepr is a sub type of object type Date representing      |
```

```
    |    a calendar entry...together with a format string             |
    |    containing the presentation description according to        |
    |    the C library function strftime()...                         |

-- TABLE OF CONTENTS

REFER Date, String;

TYPE DateRepr = Date OBJECT
                    (IN Year : INT,
                     IN Month : INT,
                     IN Day : INT,
                     IN Format : STRING)

-- ------------------------------------------------------------------------------------------
-- 3. Format representation
-- ------------------------------------------------------------------------------------------
METHOD Present : STRING;
        -- ----------------------------------------------------------------------------------
        -- Return this date formatted with its representation
        -- ----------------------------------------------------------------------------------

END OBJECT;
-- ------------------------------------------------------------------------------------------
```

Classification of 'Date':

1. Object type 'Date' under Abstraction 'TimeRepresentation'.
2. Method 'SetToActualDate' under Operation 'Set-Reset' and Operates-On 'MachineTime'. This method uses internally the Unix system call 'time()'.
3. Method 'Add' and 'Subtract' under Operation 'Add-Subtract'.

'Date' is not automatically updated to the current date or machine time. Hence 'MachineTime' was not regarded as a good abstraction for it. The methods GetYear, etc., as all others not listed above, are omitted for the simplicity of the example.

Classification of 'DateRepr':

4. (4) Method 'Present' under Operation 'Convert'.

We usually classify within one term the inverse of an operation as well, since such operations belong semantically together. The method name 'Present' denotes the application task of the method, not its function within the component. We therefore prefer the term 'Convert'.

More examples on reasoning about good terms are given in [12]. We further assume that, in a previous step, the Colibri library was assigned the Dependency terms (CooL2.1, SINIX_5.41, SNI_WX200), and 'Duration' was assigned the Abstraction 'TimeRepresentation'. The classification of built-in types of the programming language, such as integer, string, etc., is initially provided in the SIB. Figure 7.4 shows all paths through which leaf terms for the CooL Object Type 'DateRepr' are derived.

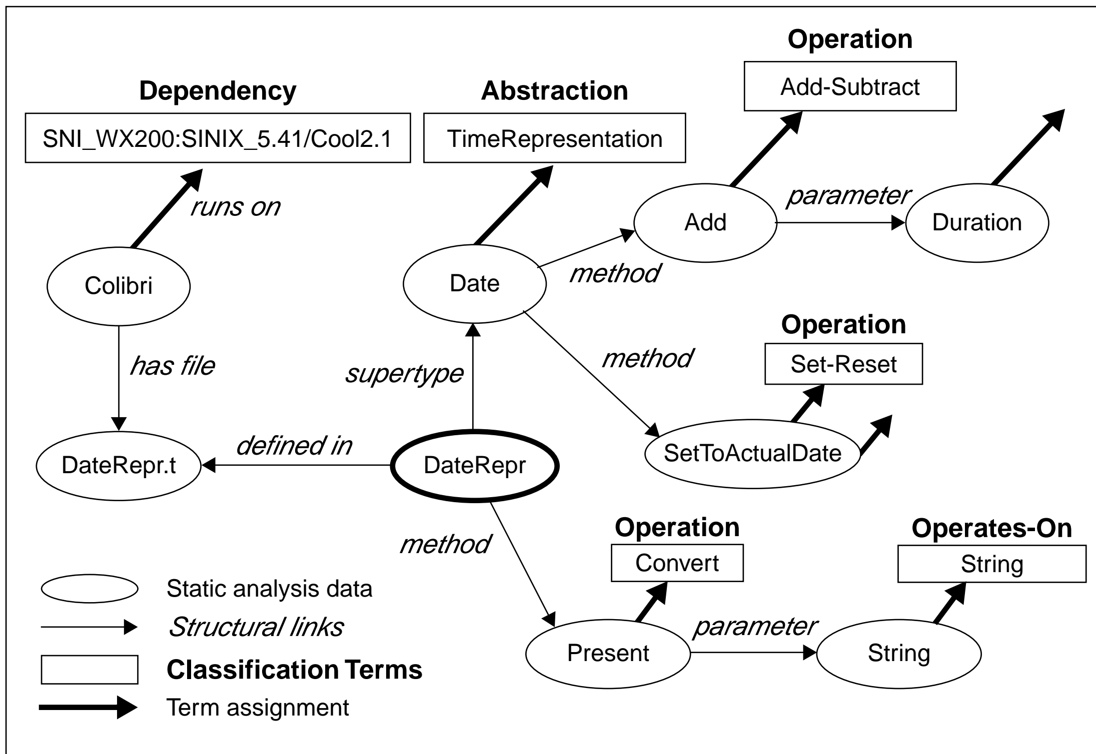

Figure 7.4 *Leaf terms valid for the CooL ObjectType 'DateRepr'.*

The complete list of terms and synonyms for 'DateRepr', resulting from the above leaf term assignment and the term space currently in the SIB, is given in table 7.1. Notice that these terms are *all* derived. Having in mind that good object-oriented applications usually derive some tens of classes from one base class, adding few methods in each derivation step, the advantage of the SIB system for classifying large class hierarchies becomes obvious.

Once found, terms are easily attached to components or correctly integrated in the term space by using the SIB facilities. To classify a class with some twenty methods we typically spend half an hour to one hour. These times, however, vary strongly with the functionality of the class. User interface classes or mathematical classes can be much more quickly classified than some internal components or components with complex functionality. For instance, characterizing the SIB query processor in contrast to other query processors is not straightforward. Evidently, the quality and maturity of existing terminology plays an important role. These observations raise interesting issues for further work, experimental as well as on the field of terminology.

Our in-house experience with the SIB classification facilities is currently based on three cases: the class library Colibri for the CooL language environment, the C++ Extended (APEX 1.0) library [1], and classes of the SIB implementation itself. The term space for the first two examples has been fully developed. The classification of the SIB implemen-

<table>
<tr><th>Facet</th><th>Derived leaf terms</th><th>Higher terms</th><th>Synonyms</th></tr>
<tr><td>Abstraction</td><td>TimeRepresentation</td><td>TemporalObject
CompoundRepresentation</td><td>Time
NonDecimalSystem</td></tr>
<tr><td>Operation</td><td>Add-Subtract
Set-Reset
Convert</td><td>StateManipulation
Arithmetic
Mathematical
Format</td><td>Algebraic</td></tr>
<tr><td>Operates-On</td><td>TimeRepresentation
MachineTime
String</td><td>TemporalObject
CompoundRepresentation
HWcomponent
List
Ordered_Collection
Collection
Bag</td><td>Time
NonDecimalSystem
CurrentTime
DateFormat
ComputerTime
Date
Calendar</td></tr>
<tr><td>Dependency</td><td>SNI_WX200:SINIX_5.41
:CooL2.1</td><td>Unix
SINIX
SINIX_5.41
SNIMachine
SNI_WX200
CooL
ObjectOrientedLanguage
CooL2.1</td><td></td></tr>
</table>

Table 7.1 *Terms and synonyms for the CooLObjectType ‘DateRepr’.*

tation is part of an on-going work to use the SIB for its complete self-documentation. Experience reports are also expected from users outside our institute.

7.7 Conclusion

The SIB classification method defines in an objective way how terms and entities will be related. This facilitates the consistent usage of the system by a group of users in that there is a high probability that two users classifying the same object will come up with the same usage of given terms, that two users will come up with the same higher–lower term ordering of given terms, and that users retrieving objects will have the same understanding of the terms as those who have classified the objects. These properties are expected to improve considerably the recall of the system. Nevertheless, there is an intellectual investment in the creation of term spaces, well known from efforts to create thesauri in other domains as well. A good term space incorporates a lot of experience and knowledge. As such, it should be subject to specific developments and exchange between user communities. In our opinion, this issue has not yet received enough attention in the literature.

Classification of software objects is a time-consuming task. We argue that the various derivation mechanisms offered in the SIB will reduce considerably the time needed for classification. They further improve the consistency of the code with the terms applied, in particular the maintenance of the applied terms after updates of the software objects. Both aspects are essential for the industrial usage of such a system.

The SIB is different from a series of other approaches in its data modelling capabilities, which allow it to integrate, without redundancies and in a single tool, the above classification mechanism with other organization principles, such as libraries, application frames, associations and lifecycle information in general. As an open, configurable system it is easily adapted to new methodologies and standards embedded into software production environments.

Integrating all aspects in a logically consistent way, as discussed above for static analysis and functional classification, gives rise to a bootstrapping and verification problem. The larger the population of the system, the more useful it is, the more important further organization principles and lifecycle information become, and the better the validity of their interconnections can be tested. To attract real users of the system, they must be provided from the very beginning with immediately useful functionalities and usage guidance. The reduction of manual work by importing as much information as possible from existing sources plays an important role in this context. The incremental development of further chains of functionality in the SIB, like the static analysis–functional classification presented here, is a main line of our future work.

References

[1] *C++ Extended Library*, APEX 1.0 Information brochure, Siemens Nixdorf Informationssysteme AG, Berlin, April 1992.

[2] *CooL V1.0*, Language Reference Manual, Siemens Nixdorf Informationssysteme AG, Berlin, April 1992.

[3] *CooL V1.0, CoLibri*, Reference Manual, Siemens Nixdorf Informationssysteme AG, Berlin, April 1992.

[4] Ralph Barletta, "An Introduction to Case-Based Reasoning," *AI Expert*, vol. 6, no. 8, Aug. 1991, pp. 42–49.

[5] Ted J. Biggerstaff and Alan J. Perlis, *Software Reusability,* Volume I: *Concepts and Models,* Volume 2: *Applications and Experience*, Addison-Wesley, Reading, Mass., 1989.

[6] Michael Brodie and Dzenan Ridjanovic, "On the Design and Specification of Database Transactions," in *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, ed. Michael Brodie, John Mylopoulos and Joachim Schmidt, Springer-Verlag, New York, 1984, pp. 277–312.

[7] Peter P.-S. Chen, "The Entity-Relationship Model: Towards a Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, no. 1, March 1976, pp. 9–36.

[8] Panos Constantopoulos, Martin Dörr and Yannis Vassiliou, "Repositories for Software Reuse: The Software Information Base," in *Proceedings IFIP WG 8.1 Conference on Information System Development Process*, Como, Sept. 1993, pp.285–307.

[9] Panos Constantopoulos, Matthias Jarke, John Mylopoulos and Yannis Vassiliou, “The Software Information Base: A Server for Reuse,” *The VLDB Journal* (to appear).

[10] Valeria de Antonellis, et al., “Ithaca Object-Oriented Methodology Manual,” ITHACA Report ITHACA.POLIMI-UDUNIV.E.8.6, Politecnico di Milano, 1992.

[11] Premkumar Devanbu, Ronald J. Brachman, Peter G. Selfridge and Bruce W. Ballard, “LaSSIE: A Knowledge-Based Software Information System,” *Communications of the ACM*, vol. 34, no. 5, May 1991, pp. 34–49.

[12] Martin Dörr and Eleni Petra, “Classifying C++ Reusable Components,” ITHACA Report ITHACA.FORTH.94.SIB.#2, Institute of Computer Science, Foundation of Research and Technology - Hellas, Jan. 1994.

[13] Pankaj Garg and Walt Scacchi, “On Designing Intelligent Hypertext Systems for Information Management in Software Engineering,” *DIF, Proceedings Hypertext ’87*, Nov. 1987, pp. 409–431.

[14] Simon Gibbs, Dennis Tsichritzis, Eduardo Casais, Oscar Nierstrasz and Xavier Pintado, “Class Management for Software Communities,” *Communications of the ACM*, vol. 33, no. 9, Sept. 1990, pp. 90–103.

[15] Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Mass., 1984.

[16] T. Hopking, C. Phillips, *Numerical Methods in Practice: Using the NAG Library*, Addison-Wesley, Reading, Mass., 1988.

[17] E. A. Karlsson, S. Sorumgard and E. Tryggeseth, “Classification of Object-Oriented Components for Reuse,” *Proceedings TOOLS 7*, Dortmund, 1992.

[18] Charles W. Krueger, “Software Reuse,” *ACM Computing Surveys*, vol.24, no.2, June 1992, pp. 131–183.

[19] Colin Low, “A Shared, Persistent Object Store,” *Proceedings ECOOP’88*, Oslo, Aug. 1988, pp. 390–410.

[20] Bertrand Meyer, *Eiffel: the Libraries*, Prentice Hall, New York, 1990.

[21] John Mylopoulos, Alex Borgida, Matthias Jarke and Manolis Koubarakis, “Telos: Representing Knowledge About Information Systems,” *ACM Transactions on Information Systems*, vol. 8, no. 4, Oct. 1990, pp. 325–362.

[22] P. Paul, “Classification of Software Components for Reuse,” SIEMENS Technical Report, July 1992.

[23] Ruben Prieto-Diaz and Peter Freeman, “Classifying Software for Reusability,” *IEEE Software*, vol. 4, no. 1, Jan.1987, pp.6–16.

[24] Ruben Prieto-Diaz, “Implementing Faceted Classification for Software Reuse,” *Communications of the ACM*, vol. 34, no. 5, May 1991, pp. 88–97.

[25] Sarada R. Ranganathan, “Prolegomena to Library Classification,” *Garden City Press*, Letchworth, Hertfordshire, 1957.

[26] Geoffrey Robinson, “UDC: A Brief Introduction,” Technical Report, International Federation of Documentation, 1979.

[27] Robert W. Schwanke, “An Intelligent Tool for Re-Engineering Software Modularity,” *Proceedings International Software Engineering Conference*, Austin, Tex., 1991, pp. 83–92.

[28] L.S. Sorumgard, G. Sindre and F. Stokke, “Experiences from Application of a Faceted Classification Scheme,” *Proceedings 2nd International Workshop on Software Reusability 1993 (REUSE’93)*, Lucca, March 1993.

[29] George Spanoudakis and Panos Constantopoulos, “Similarity for Analogical Software Reuse: A Computational Model,” *Proceedings European Conference on Artificial Intelligence*, Amsterdam, Aug. 1994.

[30] E. Svenonius, “Design of Controlled Vocabularies,” *Encyclopedia of Library and Information Science*, Marcel Dekker, New York, 1989

[31] S. Thunem and G. Sindre, "Development With and for Reuse: Guidelines from the REBOOT Project," *Proceedings ERCIM Workshop on Methods and Tools for Software Reuse*, Heraklion, Crete, Oct. 1992, pp. 2–16.

[32] Amos Tversky, "Features of Similarity," *Psychological Review*, July 1977.

[33] Costis Vezerides, "The Organization of a Software Information Base for Software Reuse by a Community of Programmers," Master's Thesis, Department of Computer Science, University of Crete, May 1992.

[34] Peter Wegner, "The Object-Oriented Classification Paradigm," in Research Directions in Object-Oriented Programming, ed. Bruce Schriver and Peter Wegner, MIT Press, Cambridge, Mass., 1987.