Managing Software Development Knowledge:

A Conceptually-Oriented Software Engineering Environment (COSEE)

By

Nagi Ghali

Thesis Submitted
to the School of Graduate Studies
in partial fulfilment of the requirements
for the Master's degree in Computer Science
under the auspices of the Ottawa-Carleton
Institute for Computer Science

UNIVERSITÉ D' OTTAWA

UNIVERSITY OF OTTAWA

ISBN 0-315-89627-2

Canada

UNIVERSITÉ D'OTTAWA
ÉCOLE DES ÉTUDES SUPÉRIEURES ET DE LA RECHERCHE

UNIVERSITY OF OTTAWA
SCHOOL OF GRADUATE STUDIES AND RESEARCH

PERMISSION DE REPRODUIRE ET DE DISTRIBUER LA THÈSE - *PERMISSION TO REPRODUCE AND DISTRIBUTE THE THESIS*

NOM DE L'AUTEUR - *NAME OF AUTHOR*
GHALI, Nagi

ADRESSE POSTALE - *MAILING ADDRESS*
127-626 King Edward Avenue,
Ottawa, Ontario K1N 9N2

GRADE - *DEGREE*: M.C.S.

ANNÉE D'OBTENTION - *YEAR GRANTED*: 1993

TITRE DE LA THÈSE - *TITLE OF THESIS*
MANAGING SOFTWARE DEVELOPMENT KNOWLEDGE: A CONCEPTUALLY-ORIENTED SOFTWARE ENGINEERING ENVIRONMENT (COSEE)

L'AUTEUR PERMET, PAR LA PRÉSENTE, LA CONSULTATION ET LE PRÊT DE CETTE THÈSE EN CONFORMITÉ AVEC LES RÈGLEMENTS ÉTABLIS PAR LE BIBLIOTHÉCAIRE EN CHEF DE L'UNIVERSITÉ D'OTTAWA. L'AUTEUR AUTORISE AUSSI L'UNIVERSITÉ D'OTTAWA, SES SUCCESSEURS ET CESSIONNAIRES, À REPRODUIRE CET EXEMPLAIRE PAR PHOTOGRAPHIE OU PHOTOCOPIE POUR FINS DE PRÊT OU DE VENTE AU PRIX COÛTANT AUX BIBLIOTHÈQUES OU AUX CHERCHEURS QUI EN FERONT LA DEMANDE.

LES DROITS DE PUBLICATION PAR TOUT AUTRE MOYEN ET POUR VENTE AU PUBLIC DEMEURERONT LA PROPRIÉTÉ DE L'AUTEUR DE LA THÈSE SOUS RÉSERVE DES RÈGLEMENTS DE L'UNIVERSITÉ D'OTTAWA EN MATIÈRE DE PUBLICATION DE THÈSES.

THE AUTHOR HEREBY PERMITS THE CONSULTATION AND THE LENDING OF THIS THESIS PURSUANT TO THE REGULATIONS ESTABLISHED BY THE CHIEF LIBRARIAN OF THE UNIVERSITY OF OTTAWA. THE AUTHOR ALSO AUTHORIZES THE UNIVERSITY OF OTTAWA, ITS SUCCESSORS AND ASSIGNEES, TO MAKE REPRODUCTIONS OF THIS COPY BY PHOTOGRAPHIC MEANS OR BY PHOTOCOPYING AND TO LEND OR SELL SUCH REPRODUCTIONS AT COST TO LIBRARIES AND TO SCHOLARS REQUESTING THEM.

THE RIGHT TO PUBLISH THE THESIS BY OTHER MEANS AND TO SELL IT TO THE PUBLIC IS RESERVED TO THE AUTHOR, SUBJECT TO THE REGULATIONS OF THE UNIVERSITY OF OTTAWA GOVERNING THE PUBLICATION OF THESES.

9/7/93
DATE

Nagi Ghali
(AUTEUR) SIGNATURE (AUTHOR)

* NB LE MASCULIN COMPREND ÉGALEMENT LE FÉMININ

FB10-AS25 88/12

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

UNIVERSITÉ D'OTTAWA UNIVERSITY OF OTTAWA

ÉCOLE DES ÉTUDES SUPÉRIEURES ET DE LA RECHERCHE

SCHOOL OF GRADUATE STUDIES AND RESEARCH

GHALI, Nagi
AUTEUR DE LA THÈSE-AUTHOR OF THESIS

M.C.S.
GRADE-DEGREE

DEPARTMENT OF COMPUTER SCIENCE
FACULTÉ, ÉCOLE, DÉPARTEMENT-FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE-TITLE OF THE THESIS

MANAGING SOFTWARE DEVELOPMENT KNOWLEDGE: A CONCEPTUALLY-ORIENTED SOFTWARE ENGINEERING ENVIRONMENT (COSEE)

D. Skuce
DIRECTEUR DE LA THÈSE-THESIS SUPERVISOR

EXAMINATEURS DE LA THÈSE-THESIS EXAMINERS

A. Mili

J. Pugh

(LE DOYEN DE L'ÉCOLE DES ÉTUDES SUPÉRIEURES ET DE LA RECHERCHE) SIGNATURE (DEAN OF THE SCHOOL OF GRADUATE STUDIES AND RESEARCH)

To my parents

Abstract

Software development, especially for large and complex systems, has long been recognized as a difficult and expensive process. Major software development problems (such as insufficient reuse of software, inadequate machine assistance for software developers, uncoordinated tools, excessive time spent during the maintenance phase, and poor documentation) have not yet been properly addressed. Most current software development environments do not provide satisfactory solutions for these problems.

In our research, we investigated these problems and we will suggest a solution that will help to eliminate some of them. We built an environment called COSEE (Conceptually-Oriented Software Engineering Environment), on top of a knowledge management system (CODE). In COSEE, we captured three most important types of knowledge needed by software developers/maintainers: domain knowledge, design knowledge, and implementation knowledge. We dynamically linked COSEE to the programming environment (Smalltalk-80) to create a unified knowledge management system for software development. We used the object-oriented approach as our design methodology and Smalltalk-80 as our implementation language. We illustrated our approach using the ATM (Automated Teller Machine) example.

Keywords: software engineering environment, software development, knowledge management system, object-oriented programming

Acknowledgments

My special thanks go to my supervisor Dr. Douglas Skuce for his support, advice and patience; without his direction, this thesis would have looked much different and would have been much less complete.

My deep thanks go to my parents for their patience and support throughout my studies; their advice and encouragement have always helped me to achieve my goals and it always will.

Finally, I would like to thank Dr. John Pugh and Dr. Ali Mili for accepting the responsibility of reviewing this work.

Table Of Contents

List of Figures

Chapter 1

Introduction

We begin by describing the major knowledge-related problems in software development, what will be our approach to solving these problems, and the organization of the thesis.

1.1 Major Knowledge-Related Problems in Software Development

It is generally agreed that developing and maintaining software, especially very large software systems, is very difficult and expensive. As Selfridge [Selfridge 90] argues: "Before attempting a particular task, a developer must often spend a great deal of time discovering features of the system, including the overall organization of the software and the location and details of specific functions and data structures". Robson et al. [Robson et al. 91] point out that: "Software maintenance is recognized as the most expensive phase of the software life cycle. The maintainer programmer is frequently presented with code with little or no supporting document, so that the understanding required to modify the program comes mainly from the code."

In developing a software system, the following major knowledge-related problems can be identified:

- Complexity of the domain
- Management of software knowledge
- Rediscovery of knowledge during the software process
- Inadequate knowledge available to the maintainer
- Insufficient reuse of software
- Inadequate machine assistance for software developers
- Poor documentation

Complexity of the Domain:

The complexity of the domain (application area) and of the task itself presents a potential challenge to the software development process. Thus, much time is spent gaining an overall understanding of the problem before beginning a specific task. Basili [Basili 90] points out that: "Most software systems are complex, and modification requires a deep understanding of the functional and non-functional requirements, the mappings of functions to system components and the interaction of components". Wirfs-Brock et al. [Wirfs-Brock et al. 90] explain also: "Software applications are complex because they model the complexity of the real world. These days, typical applications are too large and complex for any single individual to understand".

Management of Software Knowledge:

Many serious problems in software development derive from the inadequate management of software knowledge, ranging from knowledge about the programming concepts and domain knowledge to knowledge about existing software systems. Software engineers, especially novices, spend a lot of time trying to search, explore,

discover and understand software knowledge. This problem is partially due to the lack of tools and techniques for properly storing, representing, sharing and communicating knowledge; it also stems from the lack of agreement between software developers on concepts and terminology of the system under development. Communication can break down also because the special characteristics of software and the particular problems associated with its development are misunderstood. When this occurs, the problems associated with the software crisis are exacerbated sometimes causing errors due to preconceptions acquired in different academic and industrial backgrounds. Software knowledge is sometimes inconsistent or poorly represented making the software life cycle slower and longer than if it were represented using a more disciplined technique; we shall propose such a technique in this thesis.

Rediscovery of Knowledge during the Software Process:

One of the problems in software development and maintenance is that knowledge is lost during the software process. This loss of knowledge requires constant rediscovery. It happens frequently that knowledge generated in one phase is not transmitted to the next phase, adding an expensive rediscovery activity to every phase of the process.

Inadequate Knowledge available to the Maintainer:

Software maintenance consumes most of the software development process, making the overall cost of the software development very high. A major problem in the maintenance phase is that knowledge available to the maintainer is not adequate to effectively maintain the system. As Jarke [Jarke 92] points out: "Today's software systems are hard to maintain and reuse. The primary reason is their lack of integration. Although programmers have many individual tools at their disposal, there is no formal

integration across development stages, between the systems and its environments, or across development tasks". Often, software maintainers experience a lot of difficulty understanding the rationale behind an existing software system, even if it is documented. But software maintenance is critical and vital: As Wirfs-Brock et al. [Wirfs-Brock et al. 90] explain, combining a new piece of software with an existing one adds potentially numerous interactions with other pieces already in the system. Each bug that is fixed is capable of introducing numerous other bugs in seemingly unrelated parts of the system. An application can reside in a system for a long time and as it persists, it accumulates a variety of patches and makeshift accommodations. As a result, the more it gets fixed, the harder it becomes to fix it.

Insufficient Reuse of Software:

As traditional software systems evolve, developers and maintainers rarely reuse the analysis, the design, or even the code that was used earlier in the system; they reinvent the wheel. This situation is common in the software life cycle. As Bhansali et al. [Bhansali et al. 90] observe: "In development and subsequent maintenance of software systems, there are numerous occasions when a problem being solved is identical or bears a similarity to a problem that has been solved earlier". The process of solving the same problems, by repeating the same solutions or by providing other solutions, drives the cost of the software life cycle up and reduces the quality of the software.

Software reuse has been introduced primarily to circumvent this phenomenon. As Freeman [Freeman 87] points out: "the primary objective of reusable software engineering is to reduce the system-life cycle cost and improve the quality of systems. The objective includes the specific goals of reusing designs as well as code, capturing problem-domain information in a manner that facilitates its reuse, avoiding redundant work whenever possible, and amortizing

the cost of a piece of software over the largest possible number of systems".

Inadequate Machine Assistance for Software Developers:

Software developers do not get sufficient assistance from current software development tools. As Ambriola et al. [Ambriola et al. 91] point out: "In practice, people involved in developing software find the current situation frustrating, because existing tools supply only a small amount of automated assistance and tool integration".

Poor Documentation:

A major problem in software development is poor documentation. Documents can be far from accurate and may not reflect the system's main aspects. They may contain ambiguity or use terms inconsistently. They are frequently incomplete, inconsistent, or outdated. Often, developers and maintainers spend excessive time trying to understand the application domain, the design, or the implemented system, due to poor documentation. [Sametinger et al. 92] discuss the documentation problems and point out its importance on the software life cycle: "By improving the availability of complete and up-to-date documentation, we can minimize software costs considerably".

Knowledge-related problems will always affect the quality of the software as long as the knowledge management issue is not properly addressed. Hayes-Roth et al. [Hayes-Roth et al. 91] highlight the importance of knowledge engineering in software development: "Regarding software development in general, we have found the knowledge-engineering paradigm of incremental development to be highly appropriate whenever questions exist about what kind of performance is desirable, feasible, or attainable".

1.2 A Knowledge-Based Approach to Software Development

Our approach to confronting the problems described above is the following: because software development is a continuous, cooperative process of analysis and reanalysis, design and redesign, programming and program reorganization, all pertinent knowledge should be stored in a repository (i.e. a knowledge base) linked to the development tools. As Hayes-Roth et al. [Hayes-Roth et al. 91] explain the component needed in software applications: "The key difference between the new applications and more traditional ones was the need to implement and integrate knowledge-processing components; for lack of better terminology, we call such complex heterogeneous applications cooperative or intelligent systems".

Our approach will demonstrate a prototype base that offers assistance to software engineers through the use of artificial intelligence techniques. We will describe how one could use an interactive knowledge management system (CODE) for managing the different kinds of software knowledge needed during the software development process.

Although our methodology can be applied to any kind of software development, we will concentrate our discussion on *object-oriented* software development. While this is not completely general, we choose it for the following reasons:

a) Object-oriented development is becoming a widely-used technique today.
b) Our methodology is sufficiently general to be used for other non object-oriented software development as well.

In the object-oriented paradigm, "everything" is an object with states and behaviours. Object-oriented software development has been introduced as a framework that allows for a direct, natural correspondence between a model and the world, to solve some of the

problems in the software life cycle. Software development has been significantly improved by using the following object-oriented features:

- *Inheritance:* More specific objects inherit behaviour from more general ones.
- *Reuse:* Reusing software components improves the productivity of the software process.
- *Abstraction:* The representation of objects are hidden from their users.
- *Encapsulation:* Every object contains the knowledge and behaviour that are relevant to it.
- *Polymorphsim:* The ability of many objects to respond to the same message pattern.

We will manage software development knowledge by looking at software knowledge from three different points of view (*Viewpoints*) as shown in Fig 1.1: domain knowledge, design knowledge, and implementation knowledge. Our framework will store major types of knowledge needed by software developers and maintainers:

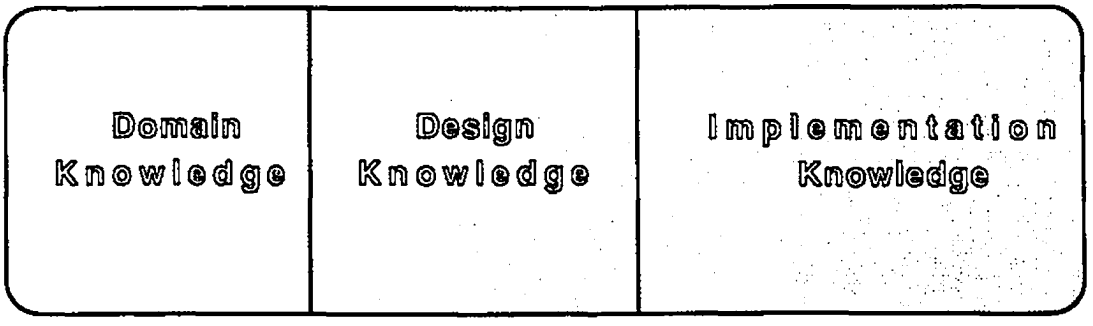

Fig 1.1 Three Viewpoints for Software Development

- *Domain knowledge* that captures all knowledge pertinent to the application domain, without any consideration of the design decisions.
- *Design knowledge* that captures all knowledge about the system design and its rationale, without requiring any dependency on the implementation language. In many situations however, the design may depend on the implementation.
- *Implementation knowledge* that captures all knowledge about the implementation of the design including the very low-level details of the implementation phase.

A software developer always needs to refer to these types of knowledge, and a maintainer should understand the overall system from all these points of views. Majidi et al. [Majidi et al. 91] argue that: "Understanding a software system requires extensive expertise and knowledge in the problem domain and in design and programming techniques."

We will show how major kinds of software development knowledge can be encoded in a machine-usable form that is also very human-usable. Our focus will be on three important phases of object-oriented development: the description (analysis) of the domain, the design of the software system, and the implementation of the design. The maintenance phase can use knowledge from any of these three levels.

We will demonstrate how to represent domain knowledge, design knowledge, and implementation knowledge in a knowledge base. These three viewpoints reflect the manner in which a system evolves from its initial description to its final implementation. *Pointers* between these viewpoints, in both directions, will help clarify the mappings from requirements and from design to implementation. In our research, we have put strong emphasis on the domain knowledge and the design knowledge, and relatively

little emphasis on the implementation knowledge.

The knowledge base will be dynamically linked to an object-oriented software development environment (Smalltalk-80) so that the developers (or maintainers) may access existing knowledge about the software component libraries.

The well-known Automated Teller Machine (ATM) example ([Rumbaugh et al. 91] and [Wirfs-Brock et al. 90]) will be discussed to illustrate the usefulness of our framework: it focuses mainly on the domain and design knowledge.

1.3 Organization of the Thesis

In Chapter 2, we describe some example of the best current software development systems that assist the software user during the development process.

In Chapter 3, we discuss various kinds of software users, knowledge representations, knowledge sources and their major problems.

In Chapter 4, we present our conceptually-oriented development approach for representing software knowledge, including the three viewpoints: domain knowledge, design knowledge, and implementation knowledge.

In Chapter 5, we describe an interactive knowledge management tool (CODE) and our approach (COSEE) to use it for software engineering. We also explain the example used, features offered by our approach for software developers, and our proposed enhancements for further research.

In Chapter 6, we offer our conclusions from the experiment and general conclusions on the relation of knowledge engineering to software engineering.

Chapter 2

Software Development Systems

Recent research on software engineering has attempted to simplify the development of software systems by providing powerful tools and sophisticated environments. However, software developers still require more assistance and guidance from more intelligent systems. A key problem, as we see it, is the lack of knowledge management facilities.

In this chapter, we describe a number of current software engineering systems: programming language environments, computer-aided software engineering (CASE) tools, knowledge-based systems (generic and software assistants), and hypertext-based systems. Of these, only the last two, as we shall explain in this chapter, are specifically intended to add substantial new knowledge management capabilities.

2.1 Programming Language Environments

Many object-oriented languages (and some conventional languages) include extensive programming environments, as well as graphical user interfaces (GUIs). These environments may include tools for browsing through existing code, writing new code, running code, debugging code, and inspecting objects. There may also be tools for tracking source code modifications in a multi-programmer environment and for analyzing space and/or time efficiency.

Smalltalk-80, Common Lisp, and C++ are considered to have the best programming language environments that provide the user with a rich environment for development and maintenance. These systems are intended to provide well-integrated support in the development of applications by single and multiple users. In addition, the languages used in these environments usually provide strong abstraction mechanisms for data and control, and include a high degree of uniformity both in the representation of objects (data structures, documents, programs, tools) and in the paradigm of interaction among different components. These environments have powerful capabilities due to the absence of software layers and the uniqueness of the implementation language: they include tools to inspect and even modify the global state of the system. Although these systems feature the extensive use of user-friendly graphical interfaces [Ambriola et al. 91], they do not feature true graphics; mainly their graphic interfaces are primarily text-oriented and not picture-oriented.

2.1.1 The Smalltalk-80 Environment

The Smalltalk-80 programming environment was written entirely in the object-oriented language Smalltalk-80, and supports the development of applications in the same language.

Programming in the Smalltalk-80 environment consists of defining new classes and methods or modifying existing system classes and methods; thus an application extends the global environment. Fig 2.1 shows a typical view of a Smalltalk-80 browser:

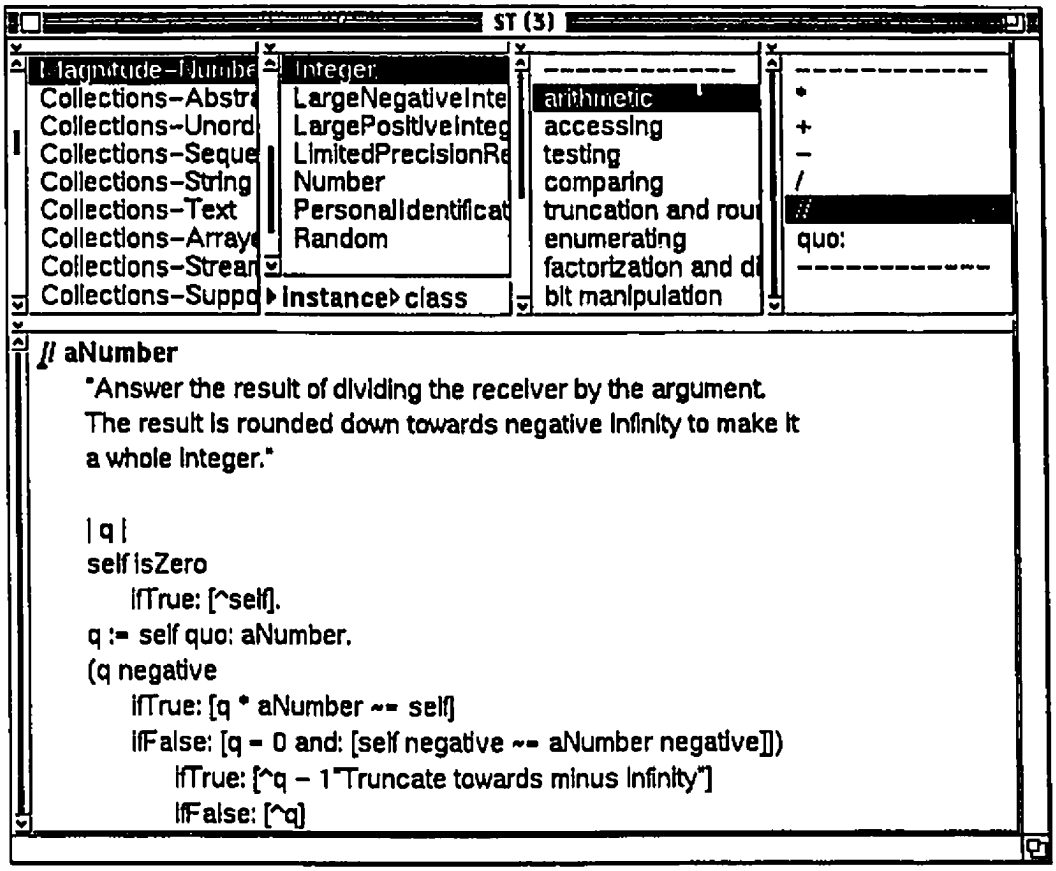

Fig 2.1 A Smalltalk-80 browser

Advantages:

- At the end of a working session, the user can save a snapshot (the state of the virtual memory: compiled methods, system objects, screen bitmap) in a file.
- The only way to share code or structure among developers is through files that contain class descriptions and method definitions.
- The graphical interface plays a key role in the system, and the extensive use of windows, menus, and mouse allows for friendly interaction with the environment. The user interface to the Smalltalk-80 system is a multipurpose interface, designed to facilitate text and graphics creation and manipulation, program development, and information storage and retrieval.

- The Smalltalk-80 environment includes several kinds of browsers, a debugger, and several inspectors. Although Smalltalk-80 has not been significantly changed since it first appeared (ten years ago), it equals or surpasses its competitors in the sophistication of its environment and the elegance of its implementation of the object-oriented paradigm. For example, Objectworks/Smalltalk's principal programming tool is the system browser. Its capabilities include not only browsing the code library, as its name suggests, but editing, compiling, and printing any selected portion of it as well.
- Smalltalk-80 environments accommodate programmers' conceptualizations of objects as independent, communicating agents by providing tools that allow them to work directly with instances. As Pugh et al. [Pugh et al. 90] explain: "Smalltalk is much more than a programming language - it is a complete program development environment. It integrates in a consistent manner such features as an editor, a compiler, a debugger, a spelling checker, print utilities, a window system, and a source code manager".
- Smalltalk-80 also provides some features that help users find classes, methods or messages-sent.
- Smalltalk-80 provides increased modularity and encourages generalization.

Shortcomings:

- The chief disadvantage from our point of view, is that some classes are difficult to understand, usually because they are inadequately documented.
- It takes several months to become familiar with Smalltalk-80, both language and system.
- As Esp [Esp 91] points out, code browsing is sometimes an uncertain and inconvenient process, involving several levels of indirection (message-sends).

- An important aspect of programming in Smalltalk-80 is finding and reusing existing classes. Experienced programmers can decide whether and how to use a given class only if they understand its purpose and its pertinent methods. As Tarumi et al. [Tarumi et al. 88] point out: "As for reusing classes, Smalltalk provides no user-friendly tools for retrieving classes. Programmers must have enough knowledge about the class library by reading manuals of each class, or by reading program codes".
- Nash et al. [Nash et al. 91] explain that Smalltalk-80 applications cannot be separated from their environment.
- The system has no ability to display any information graphically. Users can't draw any graphics without extensive programming. Newer tools can at least draw hierarchical graphs, but Smalltalk-80 does not yet have this feature.

2.1.2 The Lisp Environment

Lisp is the second oldest high-level programming language still in use, after Fortran. The major Lisp environments have most of the features of the Smalltalk-80 environment. Although the basic languages are different, there exist object-oriented extensions of Lisp; e.g. CLOS (Common Lisp Object System). We will restrict our discussion to CommonLisp since it is the only major Lisp in use today.

Advantages:

- CommonLisp provides a set of features for prototyping knowledge-intensive systems. White et al. [White et al. 89] explain that a Lisp environment contains a program interpreter, a dynamically linking loader, and a garbage collector. Because all these features are present at every point in the program development cycle, Lisp acts as its own command language, its own macro processor, and its own debugger. Such an environment differs considerably from those of the more conventional

programming languages.

- In an advanced CommonLisp environment, the loader and compiler will remember the file in which each definition appears. When asked to inspect or edit a function, the editor can find out where the function definition originated and position the editing buffer over the definition.
- CommonLisp contains a simple but effective technique for providing on-line documentation: all defining forms (such as those for variables, constants, types, macros, and functions) provide a placeholder for a user-supplied documentation string. Unlike comments, these documentation strings are part of the program and can be interrogated. For instance, the function "describe" will output the documentation string and other information associated with a symbol. Thus one can find out about symbols in a large system without having to search text files for relevant comments.
- CLOS is an interactive object-oriented system built on top of CommonLisp. Classes and methods can be defined and redefined dynamically, even while the program is running.
- CLOS is productive; it includes comprehensive standard class libraries so programmers don't have to write as much code. An unobtrusive garbage collector automatically takes care of memory management. CLOS has a complete development environment that includes integrated editors and debuggers.
- Graphical tools such as browsers and profilers help debug the code and improve performance.
- CLOS provides a great deal of uniformity. As Ambriola et al. [Ambriola et al. 91] point out: "A high degree of uniformity is achieved because every structure is a first-class object, namely it can be referred to (using pointers), passed as argument, or returned by a function. Also the interaction among objects, based on functional application, is completely uniform".

Shortcomings:

- As in Smalltalk-80, CLOS does not have any true graph-drawing capability.
- As Amriola et al. [Ambriola et al. 91] point out: "The computing model underlying Lisp is by far more complex and semantically dirty than the object-oriented one (obviously we are not considering pure Lisp)".
- CLOS does not have query-like capabilities over the class/method structure.

2.1.3 The C/C++ Environment

C/C++ also provides a software development environment with powerful features and tools. It is becoming the most popular object-oriented language environment. C++ language is built on top of the conventional C language. It is a hybrid language; Borland C++ and Microsoft C++ are the best examples of C++ environment.

Advantages:

- Borland C++ fully supports MS-Windows' advanced features, such as Object-Linking and Embedding, multimedia and true type fonts. Optimized windows allow the developer to create, edit, compile with optimization, and run Windows applications from within Windows.
- A graphical visual Object Browser allows the developer to navigate through the classes, functions or variables in the code.
- A color-coded syntax highlighter makes the code more readable and helps spot errors.
- A SpeedBar quickens windows development by employing recognizable icons to represent frequently used menu items.
- A resource workshop allows the developer to visually create a

windows user interface without programming.

- A turbo profiler helps spot bottlenecks in the code to streamline the application's performance.
- A turbo debugger provides intelligent and interactive debugging on a single monitor and a tracer helps trace windows errors.
- Application frameworks can be plugged into any developed system or can be customized.

Shortcomings:

- C++ requires the program to be complete before the developer can debug or run it. In contrast, using Smalltalk or CLOS, developers can start debugging without having written all of the lower levels of the program. They can also change one part of a program and start debugging the other parts of the program affected by the change.

Wirfs-Brock et al. [Wirfs-Brock et al. 90] explain the disadvantages of such programming language environments:

- A hybrid language provides a lot of choices - sometimes too many. Such hybrid programming code can often be harder for others to understand: for example, the same operator can represent a message-send in one context and a built-in operation in another, leading to possible confusion when others try to read the code.
- Existing data types cannot be directly extended. For example, C++ intrinsic data types such as integers and floats cannot immediately be subclassed because they are not classes. Instead, they must first be encapsulated within classes, and then a class hierarchy can be defined around them.
- C++ does not have automatic memory management; explicit language constructs (destructors) allow programmers to specify what will happen when an object is deallocated. Explicitly finding and destroying unused objects can be a tedious, time consuming and frequently error-prone process.

2.2 Computer Aided Software Engineering (CASE) Environments

The term CASE is defined, broadly, as the tools and methods that support an engineering approach to software development at all stages of the process.

CASE has been successful in focusing attention on the need to establish software development as an engineering discipline. The fundamental rationale for the increase in the use of CASE tools in the industry is the belief that CASE tools facilitate and enhance productivity and system quality. The development of CASE environments has evolved over several years. As Urban [Urban 92] explains, "users are demanding high level, domain-specific interfaces to applications, easy-to-use systems, systems that offer increased productivity/cost ratios and systems that are modular, portable, and robust".

Advantages:

- By automating many of the more routine software development tasks and performing automatic transformations between representations, CASE has demonstrated an ability to boost productivity and prevent defects.
- Advanced CASE tools are making it more feasible to introduce semiformal and formal methods to the development process by removing clerical overhead and enforcing rigourous design rule checking.
- Norman et al. [Norman et al. 92] explain: "A CASE environment lets systems developers document and model an information system from its initial user requirements through design and implementation and lets them apply tests for consistency, completeness, and conformance to standards". It provides the system developer with facilities for drawing a system's architecture diagrams, describing and defining functional and data

objects, identifying relationships between system components, and providing annotations to aid project management. The user's various work products are stored in an integrated, non redundant form in a central repository or dictionary either on the workstation or on a central server or host system. The system definition as a whole can be checked for consistency and completeness. Analysis can be performed on the information collected or defined to date, thus supporting incremental development and the detection of inconsistencies and errors early in the life cycle. The documentation required by organizational or deliverable standards can be generated from the system description in the dictionary. Also, generators for database schemas and program code are being incorporated in, or interfaced to, CASE environments to provide a step toward automated system generation.

- A comprehensive CASE development environment for the front end of the life cycle integrates several component tools and facilities. The system developer can work on diagrams such as dataflow diagrams, structure charts, entity-relation diagrams, logical data models, presentation graphs, state-transition diagrams, transformation graphs, and decision matrices. The user can directly create diagrams for system documentation. Analysis facilitates check for consistency and completeness. End-users screens and reports can be developed for the system under design. Deliverable documentation can be organized graphically and can incorporate diagrams and text from the central dictionary.
- Besides serving as an aid to productivity which helps to capture system-design knowledge, a CASE environment provides new opportunities for using analysis techniques to improve some aspects (such as reliability and efficiency) of information systems before they are implemented. It can also help verify a completed system against its design and maintain the system description as accurate documentation.

Shortcomings:

Even so, CASE is not a satisfactory software assistant. Although CASE has significantly influenced the practice of system development, its potential is limited by the difficulties involved in integrating tools in a cohesive environment:

- Among the greatest challenges is the need for tighter integration among tools in a manner that supports openness to a variety of methods, notations, processes, tools, and platforms.
- Understanding the software process and getting developers to use software engineering techniques correctly and consistently will remain a problem, especially in the face of evolving technology.
- Forte et al. [Forte et al. 92] explain that while CASE has already achieved substantial success in defect prevention, we are reaching a plateau due to the limits of our knowledge about the software development process. Areas that are particularly weak in process definition are requirements elicitation, software maintenance, re-engineering, and object-oriented techniques.
- Available CASE tools address only a portion of the maintenance activity and are not well integrated with tools for new development. It is acknowledged that CASE integration standards are not mature and will continue to evolve in the future.
- Current CASE technology still encourages an individual approach to development. A serious shortcoming is the lack of support for the communication between developers and end-users and among developers themselves. CASE environments do not incorporate collaborative tools (groupware) to support cooperative development.
- Potential CASE users are looking for open environments spanning life cycle stages, development roles, distributed networks, multivendor tools and computing platforms.
- CASE tools are still weak in reusing software components. As Norman [Norman 91] explains: "Current CASE technology does not provide adequate support for software reuse in terms of

classification, selection, understanding, modification and adaptability".

- Lowry [Lowry 91] explains: "The current generation of case tools are limited by shallow representations and shallow reasoning methods. CASE tools will either evolve into or be replaced by tools with deeper representations and more sophisticated reasoning methods. The enabling technology will come from AI, formal methods, programming language theory, and other areas of CS".

TurboCase 4.0

TurboCase 4.0 is good example of a CASE tool. It supports object-oriented analysis by adding behaviour modelling to the entity relationship diagram. It supports object-oriented design with four diagram types: Class hierarchy, Class Collaboration, Class Definition, and Class Design diagrams. A data dictionary and checking rules are linked to these diagrams. Also, TurboCase 4.0 integrates structured analysis and techniques within its environment.

ObjecTime

ObjecTime [Selic et al. 92] is another example of an object-oriented CASE tool that is targeted for event-driven systems, including those with a high degree of complexity and distribution. It enables the creation of executable analysis and design models. It covers a broad spectrum of application, from system architecture and protocol verification to detailed software design and implementation. It supports the Real-time Object-Oriented Modelling (ROOM) methodology, encouraging iterative development. Graphically-captured designs are executed and validated in an extensive integrated run-time environment. The high-level design paradigms supporting real-time include concurrent objects (hierarchically decomposed) that communicate via messages through formal protocol definitions. Complex hierarchical finite-state machines specify the behaviour of such objects. Inheritance can be applied at the design component level, independent of the detail level programming language.

2.3 Knowledge-Based Systems

The difficulty in constructing and maintaining large software systems based on existing technology has become widely recognized [Selfridge 90]. A primary challenge consists in the need to maintain up-to-date knowledge about a complex and evolving system.

The key problems that arise when designing large software systems are how to organize a large amount of disparate knowledge and how to acquire, maintain and extend that knowledge when appropriate.

Since the early 80s, researchers have been investigating the notion of using knowledge-based systems to manage large software systems (such as [Waters 81], [Green et al. 83] and [Neighbors 84]).

First developed more than two decades ago in artificial intelligence research, knowledge-based systems have seen widespread application in recent years. While performance has largely been the focus of attention, building such systems has also expanded our conception of a computer program from a black box providing an answer to an "open" system capable of explaining its answers, acquiring new knowledge, and transferring knowledge to users. These abilities derive from the clear distinction between what the program knows and how that knowledge will be used, making it possible to reuse the knowledge in different ways.

In this section we discuss two kinds of knowledge-based systems: generic knowledge-based systems and knowledge-based software assistants (KBSA).

2.3.1 Generic Knowledge-Based Systems

Generic knowledge-based systems are knowledge-based systems that can be used for any kind of knowledge; they are not

dedicated to a particular application. These systems share a common goal which is manage (enter, edit, store, retrieve, etc.) knowledge in a knowledge base, but they differ in the features they provide to the user and the inferences they can perform. A number of knowledge-based systems have appeared during the past fifteen years. We selected three different kinds of knowledge-based systems as illustrative examples and we will explain how they differ in their purposes.

2.3.1.1 CODE

CODE [Skuce et al. 92] is a knowledge management system specifically designed to meet the needs of interactive knowledge management.

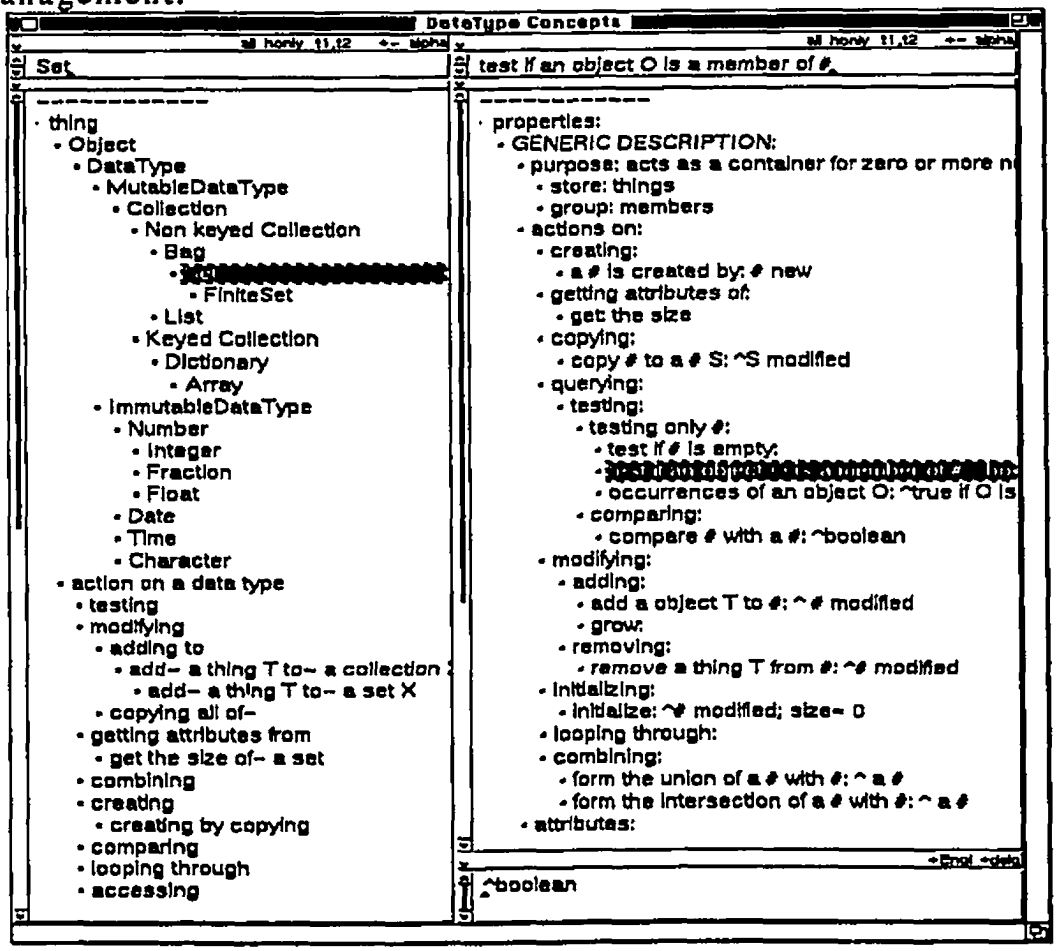

Fig 2.2 A CODE browser: Understanding prog. lang. concepts

Advantages:

- As stated in [Skuce et al. 92], CODE combines some of the most useful features of frame-based inheritance systems and conceptual graphs, favouring expressiveness over the ability to perform complex automatic inferencing.
- CODE particularly focuses on assisting users to formulate and analyze concepts and word meanings, and to retrieve relevant knowledge.
- It can be used for intensive human interaction like design, documentation and tutoring. The prime aspects of designing CODE are its ease of implementation and its ability to help users organize their ideas in a simple and flexible way. The emphasis is on allowing the user to do the required inferences freely and easily, on the support for language-related problems, and on flexible user interface facilities for locating and viewing knowledge.
- While attention has been given to technical knowledge management, a level of generality has been maintained to support knowledge management in any subject area in which concepts can be reasonably and precisely described. It can be used as an assistant for prototyping and knowledge experimentation.
- In CODE, the degree of formality can be varied according to the user's preferences. Knowledge can be a mixture of very informal (unstructured natural language) or highly formal (expressed in some version of logic); the greater the level of formality, the greater the system's ability to perform syntax and semantic checking. The user can sketch knowledge rapidly and later make it incrementally more correct and formal. The CODE designers sought a middle ground between systems that were too simple to capture a wide variety of knowledge and those that were too complex, preventing most users from being able to understand their syntax and semantics.
- To represent a statement in CODE, the user must specify a thing (the subject) in the "is-a" hierarchy and a predicate in the

predicate hierarchy. Facet predicates allow the user to make incremental additions to any statement. CODE supports facet-level inheritance; i.e. not all facets may inherit. It permits users to treat predicates as concepts, i.e. to make statements about predicates.

- The main inferencing capabilities in CODE are: inheritance, delegation, and an off-line full first logic order system (FOLDE) to perform forward/backward chaining, contradiction and inconsistency checkings and semantic errors detection. A natural language parser (ClearTalk) compiles rules expressed by the user into a format used by CODE, FOLDE, or another system.

Shortcomings:

- Lacks support for natural language dialogue systems.
- Weak knowledge-base partitioning capability.
- Lacks rule-based inferencing.
- Lacks critiquing of semantic errors.

We do not describe CODE in detail in this section because it is described further in chapter 5. In this way, the description of our approach, COSEE, which uses CODE, will be self contained in chapter 5.

2.3.1.2 CYC

Cyc [Lenat et al. 90] is a very large, controversial, frame-based system used to encode a large amount of common-sense knowledge that people intuitively use to understand the world. This, it is hoped, would permit computers to be able to process knowledge, e.g. from natural language documents, that they otherwise could not use.

Advantages:

- The knowledge base is intended to overcome the brittleness and knowledge acquisition bottlenecks encountered in current software

systems.

- It will also support expert systems, natural language systems and other artificial intelligence systems. As Lenat et al. [Lenat et al. 90] explain, the rationale is that today's programs do not really understand natural language very well, they do not have general knowledge from which to draw conclusions and they do not have far-flung knowledge to use for comparisons; they are not equipped to dynamically grapple with a situation when it exceeds their current limitations.
- The goal of designing the Cyc representation language CycL is to allow users to interact with the system at an epistemological level as well as at an heuristic level. The epistemological level uses a language that is essentially a first-order predicate calculus with augmentation for reification (i.e. having a name for propositions, and being able to make statements about other statements) and reflection (e.g. being able to refer to the facts supporting the system's beliefs in another fact in axioms). The heuristic level, by contrast, uses a variety of special purpose representations and procedures for speedy inference. The heuristic level is a "compilation" of the epistemological level. This approach leads to the existence of the knowledge base at two levels: the epistemological level and the heuristic level, and the user can interact with CycL at either of these levels.
- Particular emphasis is placed on a large built-in ontology.
- Cyc is designed to act as an automatically as possible since it is designed to answer user questions entirely on its own.
- Cyc has the ability to make many automatic inferences including forward/backward chaining.
- Cyc uses a frame representation with a variety of slot types representing properties. These slots have the same structure and always inherit as a whole.

Shortcomings:

- Built-in ontology is complex and controversial. Only the designers have accepted it.
- Great emphasis is placed on automatic inferencing at the expense of user expressiveness.
- Lack of a good user interface system. As Skuce [Skuce 92] points out: "a system like Cyc is an extreme example of one needing a good user interface". (A user interface undoubtedly exists, but it has never been described).

2.3.1.3 SB-ONE

SB-ONE [Kobsa 91] is a knowledge representation workbench for representing conceptual knowledge, with the emphasis on applications in natural language systems. SB-ONE belongs to the KL-ONE family that uses automatic classification as its main inference mechanism [MacGregor 91].

Advantages:

- Special emphasis is put on supporting the knowledge engineer in building, browsing, and correcting knowledge bases for natural language dialogue systems.
- Kobsa [Kobsa 91] explains that besides the SB-ONE language, the workbench comprises three different interfaces (functional, textual, and graphical), a partition mechanism, a consistency maintenance system for the syntactic well-formedness of SB-ONE knowledge-bases, a classifier, a realizer, a pattern matcher, a spreading activation mechanism, an interpreter and classifier for SB-ONE to SB-ONE translation rules, an integration mechanism for an external frame-based representation, and a connection between SB-ONE and an extended Prolog.
- The knowledge representation language handles knowledge at

three levels: The epistemological level (first order predicate rules help explain how well-formed knowledge representation expressions can be formed from knowledge representation elements) , the interpretational level (relates knowledge representation expressions and elements to the domain), and the notational level (through the use of graphical and linear notations).

- A TELL/ASK facility accepts the knowledge in textual form with some constraints and permits the user to do queries.

Shortcomings:

- The main inferencing mechanism is automatic classification, which is based on the assumption that given a hierarchy of definitions, a new definition can be classified in this hierarchy according to its properties. This mechanism, more than in Cyc, limits the expressiveness of the users.
- The knowledge unit is a general concept that consists of a concept predicate, concept name, set of attribute descriptions and concept types. An attribute description also forms another structure. SB-ONE has many complex relations between concepts and attribute descriptions. This structure and these relations make the system hard for users to learn and to use.

2.3.2 Knowledge-Based Software Assistants (KBSA)

So-called "Knowledge-based software assistants" (KBSA) are systems developed mainly for managing knowledge about software in a knowledge base. In 1983 RADC (Rome Air Development Conference) published a report [Green et al. 83] calling for the development of a knowledge-based software assistant, which could employ artificial intelligence techniques to support all phases of the software development process. Since then, an annual KBSA conference has been held to provide a forum for discussions and presentation of work related to the KBSA effort. KBSA provide a

promising and serious approach for addressing software knowledge management problems. KBSA is a proposed architecture to aid the development, evolution, and maintenance of large software projects. Software development and maintenance under the KBSA paradigm is fundamentally different from current software engineering practice; changes are made at any level of the system (e.g. the requirements, the specifications, the design) rather than just to the software itself. Also, KBSA captures design rationale and can act as an intelligent software assistant to developers, maintainers, and end-users.

There are different kinds of KBSA based on different criteria; e.g. transformation, formality, language-specific. We will discuss these three kinds of KBSA:

1) Transformational programming (or automatic programming) attempts to develop and maintain software systems at the specification level and automatically transform it into production-quality software. This process is achieved with the help of knowledge-based tools. An example of this kind of KBSA is KIDS [Smith 90] in which users interactively perform correctness-preserving transformations on a formal specification in order to produce an efficient implementation. The programming is carried out at a very abstract level: the user describes which algorithmic clichés to apply, such as simplification or finite differencing (adding data storage to a function to prevent unnecessary recalculation). The final step, inference of the actual implementation, is automatic. Other systems do not take the approach of transformational programming; rather they prefer semi-automatic assistance (e.g. The Programmer's Apprentice [Rich et al. 89]). Their current approach represents a major change: they started their project with the long term goal of automating the programming process, but they later changed its emphasis to that of building an intelligent assistant for expert programmers, with more emphasis on the requirements.

2) KBSA that use formal specification methods provide a mathematical basis for statements made about software. The

primary goal of formalizing specifications is to improve understanding of what must be implemented, thereby reducing implementation errors and maintenance. A major benefit of formal methods is that they make unambiguity possible, but they involve human creativity in producing mathematical foundations for this formality. ARIES, described by [Johnson et al. 91], is an example of a KBSA that uses formal specification.

3) The Knowledge-Intensive Development Environment described by [Schoen et al. 88] is a kind of KBSA that uses specific language to assist in the development of software systems. It uses an object-oriented language called Strobe, a lisp-based, object-oriented programming language. Strobe is useful in two ways: 1) as a programming paradigm, it is the link that joins distinct software subsystems in a uniform manner; 2) as a simple representation language kernel, it supports the construction of computational models which mirror the organization of the physical world in which the software systems are to operate.

The goals of Knowledge-based software assistants and CASE tools are similar, and the terminology they use is often the same. The main difference is that KBSA are derived from artificial intelligence research while CASE tools come from software engineering research. CASE is product-oriented while KBSA is process-oriented; CASE is well-engineered (its greatest strength) while KBSA is a laboratory prototype; and finally CASE is team-oriented while KBSA is individual-oriented. The central role of many KBSA is transformations; its emphasis on formalism and existing process orientation are important distinguishing differences. CASE basically exist only in environments where software engineering is performed, and does not use formal specifications.

Advantages:

- It acts as an intelligent assistant (both reactive and proactive) to formally derive code.

- It contains a knowledge base in which the user encodes the system knowledge and its derivation histories.
- Formal specification can act as a working prototype.
- Systems would be developed through evolutionary transformations.
- The reuse of knowledge. In software development, the concept of reuse is growing in popularity, thus methods must be devised to reuse knowledge more effectively. A knowledge-based approach is linked to reuse and is also needed to manage/coordinate knowledge within a project.
- The knowledge-based approach allows for the recovery of knowledge about the system once software developers are gone. It also attempts to retain the know-how of software production; in so far as the concepts used by software designers and the knowledge of programmers can be formalized, software design and implementation becomes a process that is in itself recordable, analyzable, reusable, and to some degree, automatable.

Shortcomings:

- Does not provide a comprehensive approach to software development.
- Does not help sufficiently in a team environment.
- Need to include extensive graph diagrams capabilities.
- Cannot generate a complete documentation for the developed system.

2.3.2.1 LaSSIE

We next discuss an example of a knowledge-based system for software that is based on a generic knowledge representation, "KL-ONE" described in [MacGregor 91]. It bears more close comparison to

our approach than the earlier systems.

The LaSSIE system is a prototype that uses a frame-based description language and makes inferences based on its classification hierarchy. LaSSIE is an attempt to attack the problems of invisibility (structure of software is hidden) and complexity of software systems. This approach applies an existing knowledge representation and reasoning system to the management of information about large systems. The primary motivation is the need for accessing up-to-date information about a complex and evolving system. Devanbu et al. [Devanbu 90 et al.] argue that the main problem of large software systems is the discovery problem, i.e., the problem of learning about (understanding) an existing system in order to use or modify it.

The LaSSIE knowledge base primarily describes the functioning of the software system from a conceptual viewpoint, with some information about its architectural aspects. This knowledge base is intended to help prevent the loss of architecture knowledge by explicitly codifying the primitives supported by the architecture into a formal, taxonomic knowledge base and making it available for browsing and querying. The LaSSIE's knowledge base proposed by [Devanbu et al. 91] contains only action concept descriptions classified into a conceptual hierarchy. Query processing is carried out in two stages: First, the query is placed in LaSSIE taxonomy by the classification algorithm, using the description of the query and descriptions of the frames in the taxonomy; then, the matching instances are the instances of those frames that are subsumed by the classified query. These are considered to be the answer. LaSSIE has a natural language interface that maintains data structures for each of several types of knowledge. This information includes: a taxonomy of the domain (which enables the parser to perform several types of disambiguation), a lexicon (which lists each word known to the system along with information about it), and a list of compatibility tuples (which indicate plausible associations among objects and thus reflect the semantics of the domain).

The knowledge base is built using a classification-based knowledge representation language, KANDOR (a member of the KL-ONE family), to provide semantic retrieval. Besides serving as a repository of information about the system, the knowledge base serves as an intelligent index for reusable components. In KANDOR, a frame is considered a complex description which expresses constraints on members of the class that it denotes. The restrictions in a frame definition are usually specified in terms of slots, which are two-place relations that describe the attributes of class members. Restrictions can be formed by limiting the type of slot-filler expected or by specifying the maximum and minimum number of fillers expected. The values of the slots are concepts from the taxonomy. KANDOR performs two kinds of inferences: inheritance of properties and automatic classification.

Advantages:

- Addresses the problems of invisibility and complexity of large software systems.
- Captures the functionality and the architectural aspects of the software systems into a conceptual hierarchy in a knowledge base.
- Has a natural language interface to maintain the data structures of several types of knowledge.
- Its knowledge base serves as an intelligent index for reusable components.

Shortcomings:

Devanbu et al. [Devanbu et al. 91] argue that the limitations of LaSSIE are mainly the limitations of KANDOR. These limitations are:

- KANDOR is a domain-independent language, not specifically designed to represent knowledge about real-time software in terms of objects and actions. Thus, there are various aspects of each that cannot be expressed adequately within its representation framework. (e.g. KANDOR does not support reasoning based on

part-of hierarchies).

- KANDOR seriously limits the expressiveness to make the classification algorithm faster and easier to implement.

2.3.2.2 CODE-BASE

CODE-BASE [Selfridge 90] is a software information system that uses frame-based knowledge representations to represent a wide spectrum of knowledge about telecommunications software. It uses several techniques to ensure that the knowledge base is synchronized with the code. While LaSSIE attacks the problem of invisibility and discovery at a higher level domain, CODE-BASE tries to solve this problem by linking the domain knowledge to the code itself. CODE-BASE represents a description of C code and generic Unix information. Selfridge [Selfridge 90] explains: "the user queries CODE-BASE in a query language; then, from CODE-BASE, he/she receives a list of matching instances. The user can create new concepts or categories and populate them from the results of a query, as well as create combinations of old concepts. These new concepts can then be used in subsequent queries".

CODE-BASE is built on top of Classic (a member of the KL-ONE family) which provides the following kinds of inferencing: inheritance, classification, contradiction detection and simple forward chaining. It includes two kinds of automatic classification: classification of concepts and classification of individuals. Classification of concepts takes a new concept description and automatically places it in the proper part of the taxonomy. Classification of individuals is similar: given a new individual, Classic will determine the concepts that that individual is an instance of. Concepts are stored in a taxonomy which represents an is-a hierarchy and provides for the object-oriented inheritance of concept properties. For knowledge about which individuals are instances of a particular concept, each concept in the hierarchy has an associated

"meta-concept" that represents the number of individuals that are instances of that concept.

CODE-BASE is intended for the reverse-engineering of existing large systems. It concentrates on representing the code knowledge that can be extracted automatically. A querying mechanism uses the code information that is stored in a database and loaded on demand. There are three types of code knowledge that are represented in CODE-BASE. The first is the file and directory structure of the software base for the telecommunications system. The second is the definition and use of code objects, including files, functions, macros, type declarations, and global variables. The third is the set of processes that make up the software system and the set of messages between these processes.

The knowledge acquisition in CODE-BASE is done automatically through systems that extract from C source files the code objects, their relations with other objects and the places where they are used. These code objects are then represented in an is-a hierarchy with inheritance.

Knowledge retrieval is done by typing a query that has to follow a specific syntax. This query forms a new concept, and populates the concept with all functions defined in files and matching a certain string. The user then uses the browsing ability in the interface to examine each function and discovers that a certain function is the primarily function for the query. The user repeats this process iteratively until the required function is discovered.

Advantages:

- Attempts to solve the problems of invisibility and discovery by statistically linking the domain knowledge to the code itself. It is primarily intended for reverse engineering of existing large software systems.
- Captures in its knowledge base descriptions about the C code and

generic Unix information; i.e. syntactic code knowledge about the system.

- Extracts the knowledge from the code automatically in the form of objects and represents them in an is-a hierarchy.
- Uses several kinds of inferencing (such as inheritance of property, contradiction detection, and simple forward chaining) and automatic classification.

Shortcomings:

- No dynamic link between the domain knowledge and the code itself.
- Automatic classification of the Classic system limits the expressiveness in the knowledge acquisition process.
- Knowledge retrieval process presents some complexity and difficulty for the user.

2.3.2.3 The Programmer's Apprentice Project

The Programmer's Apprentice project deals with three main phases of software development. The project itself is considered to have three phases. Since the project designers started with the implementation first, these three phases are: implementation, design, and requirements. As Rich et al. [Rich et al. 89] explain, the long term goal of this project is to develop a theory of how expert programmers analyze, synthesize, modify, explain, specify, verify, and document programs. The two basic principles underlying this project are: the assistant approach and inspection methods. In general, a cliché consists of roles (i.e. properties) and constraints (which are used to specify fixed elements of structure, to verify the parts that fill the roles, and to compute how to fill empty roles).

The Programmer's Apprentice (PA):

The first phase for the project team was, chronologically, the implementation phase. An implementation cliché mainly captures knowledge about the implementation using primary and described roles, comments, and constraints. The key to achieving success lies in the shared programming knowledge that makes the communication between programmers possible. The inspection methods are based on the premise that, given a library of clichés, it is possible to perform many programming tasks by inspection rather by reasoning from first principles. The Programmer's Apprentice focuses on the use of inspection methods to automate programming. Codifying clichés is a central activity in this project. The Plan Calculus is used as a formal representation for programs and programming clichés. A Plan Calculus is essentially a hierarchical graph structure made up of different kinds of boxes (denoting operations and tests) and arrows (denoting control and data flow). It combines representation properties of flowcharts, dataflow schemas, and abstract data types. A related system called Cake consists of a knowledge representation component and a reasoning component. Cake combines special purpose representations, such as frames and the Plan Calculus, with general purpose logical and mathematical reasoning.

KBEmacs (knowledge-based editor in Emacs) is a prototype of a part of the PA developed to demonstrate the usefulness of the assistant approach and of clichés in the implementation part of the software process. Two main tasks in the development of a prototype KBEmacs are: automatic generation of program documentation (explaining the program in terms of the clichés used) and programming language independence. KBEmacs can automatically implement a program once a software engineer has selected the appropriate algorithmic fragments to use. KBEmacs supports both retrieval of clichés and reuse. Implementation clichés include knowledge about the program itself (such as file names used, input, output). A drawback of KBEmacs is that the user must know the clichés in the library by name to retrieve them. However, all systems require the user to use exactly the terms known to the system.

The Design Apprentice (DA):

The second phase of the project is the Design Apprentice (DA) which is a tool that can assist a programmer in the detailed design of programs. [Tan 89] explains that the tool supports software reuse through a library of commonly-used algorithmic fragments, or clichés, that codify standard programming. The cliché library enables the programmer to describe the design of a program concisely. Design clichés include knowledge about the specifications, design, and hardware. Each of these clichés is annotated with information about what roles and constraints are mandatory, likely, or possible. The DA can detect some kinds of inconsistencies and incompleteness in program descriptions. It automates detailed design by automatically selecting appropriate algorithms and data structures. It supports the evolution of program designs by keeping explicit dependencies between the design decisions made.

The Requirements Apprentice (RA):

The third phase of the project is the Requirements Apprentice (RA) which assists a human analyst in the creation and modification of software requirements. Reubenstein et al. [Reubenstein et al. 91] explain that unlike most other requirements analysis tools, which start with a formal description language, the focus of the RA is on the transition between informal and formal specifications. A major problem that faces the RA is knowledge acquisition. The RA supports the earliest phases of creating a requirement, in which ambiguity, contradiction, and incompleteness are inevitable. It attempts to overcome the problems of human communication, especially abbreviation, ambiguity, poor ordering, contradiction, incompleteness and inaccuracy. The RA accepts a restrictive natural language input and produces three kinds of output: interactive output (that notifies the analyst of conclusions drawn and inconsistencies detected while requirements information is being entered), a machine Requirements Knowledge-Base RKB (that represents everything the RA knows about an evolving requirement), and a Requirements Document (that resembles a traditional requirements document summarizing the RKB). The RA is composed of three modules: a knowledge-

representation and reasoning system (Cake), an executive that handles interaction with the analyst and provides high-level control of the reasoning performed by Cake, and a cliché library which acts as a repository of information relevant to requirements in general and to domains of particular interest. Compared with implementation and design clichés, the range of clichés involved in software requirements is much more open-ended. Any part of the real world may be relevant in specifying a requirement. In a given application, the Apprentice will be useful to the extent that the relevant clichés have been codified.

Advantages:

- Captures three important types of software development knowledge in the form of clichés: the implementation knowledge, the design knowledge, and the requirements knowledge.
- The Programmer's Apprentice (PA) helps in representing programs and programming clichés, in knowledge representation and reasoning process, and in automatic generation of program documentation.
- The Design Apprentice (DA) helps in reusing design components and in capturing design rationale. It can also perform some kinds of inferencing (such as inconsistency detection).
- The Requirements Apprentice (RA) helps in the transition stage from the informal to formal descriptions of the domain.

Shortcomings:

- All three phases require the user to use exactly the same names and terms known to the system in the process of knowledge retrieving.
- Codifying clichés is a complex task for the user to perform at all three levels of the project.
- No good user-friendly interface in knowledge acquisition/retrieval.

2.4 Hypertext Systems

Hypertext systems are systems which allow direct machine-supported references from one textual or iconic unit to another; they enable the user to interact directly with these chunks and to establish new relationships between them. Conklin [Conklin 87] explains the concept of Hypertext: windows on the screen are associated with objects in a database, and links are provided between these objects, both graphically (as labeled tokens) and in the database (as pointers). Nielsen [Nielsen 90] defines hypertext as a non-sequential writing: a directed graph, where each node contains some amount of text or other information and the nodes are connected by directed links. He also explains that hypertext can be perceived as a computer-based medium for thinking and communication that extends conventional linear documentation.

Advantages:

Conklin [Conklin 87] explains that the advantages of hypertext are:

- Supports structuring
- Features the modularity and encourages consistency of information
- Allows for the customization of documents
- Provides global and local viewing of documents
- Allows for task stacking
- Enables collaboration between users

Shortcomings:

- Conklin et al. [Conklin et al. 89] highlight two major problems with hypertext: the disorientation problem (the tendency to lose one's sense of location and direction in a nonlinear document) and the cognitive overhead (the additional effort and concentration necessary to maintain several tasks or trails at one time).

- Lucarella [Lucarella 90] focuses on one major shortcoming of hypertext systems which is information retrieval. In this context, the retrieval process is regarded as a process of inference that can be carried out either by the user exploring the hypertext network (browsing), or by the system, exploiting the hypertext network as a knowledge base (searching). A comprehensive model should take into account both of the perspectives, effectively combining browsing and searching in a unified framework.
- Smeaton [Smeaton 91] summarizes the main issues and problems in retrieving information from hypertext: hypertext uses a browsing strategy rather than a searching strategy, thereby reducing the freedom hypertext gives to users in choosing the information they wish to see.

gIBIS

gIBIS is a hypertext tool that provides a clear and natural structure for a discussion or a deliberation process. Conklin et al. [Conklin et al. 89] explain that the goal of gIBIS is to facilitate and capture policy and design discussions. It implements a specific method, called Issue Based Information Systems (IBIS) which was developed for use on large, complex design problems while capturing the design rationale with little disruption of the normal process. gIBIS makes use of colour graphics and a high speed relational database server to facilitate building and browsing typed IBIS networks. It is designed to support collaborative construction of these networks by any number of cooperating team members spread across a local area network.

Motivations for gIBIS are the capture of the design rationale, the support of computer mediated teamwork, and the need for an application with a large information base that can be used to investigate and navigate through very large information spaces.

gIBIS can be perceived as a hypertext system with prescribed semantic types; the IBIS method imposes a limited selection of node

and link types on the user. The tool does not provide the user with a brainstorming feature; rather the IBIS method requires structured materials.

Advantages:

- Intended for capturing the design rationale of complex design problems.
- Provides strong browsing capabilities.
- Supports collaborative work between designers; i.e. can be considered as a groupware.

Shortcomings:

- Lacks support for brainstorming capabilities.
- Cognitive overhead is noticeable: the freedom of choice, inherent in branching documents (in a network of nodes), simply requires substantial care from the writer and considerable attention from the reader.

2.5 Summary and Relevance to Our Work

Our goal is similar to the software systems (all types) discussed in this chapter: to provide an environment for software development.

Compared to specifically other KBSA systems (LaSSIE, CODE-BASE, and The Programmer's Apprentice), we share the same goals:

- Use a knowledge-based system for software development
- Encode software knowledge in a knowledge base in an organized and a structured way:
    - LaSSIE describes the functional and architecture aspects
    - CODE-BASE describes the implementation knowledge

- The Programmer's Apprentice describes requirements, design, and implementation knowledge

However, all software systems still weak in:

- Providing a wide assortment of knowledge management capabilities
- Providing support for natural-language related problems
- Relying heavily on external documentation
- Having limited scope; designed to perform specific functions
- Being uncoordinated; no link between them

In particular, most KBSA have the following shortcomings:

- Strongly limits user expressiveness due to the use of knowledge representation systems designed to support automatic inferencing
- No good user friendly interface in either the knowledge acquisition or the knowledge retrieval (complex and iterative process)
- No links between the knowledge base and the programming environment
- Requires the user to enter queries using exactly the same names and terms known to the system in the knowledge retrieval process

Chapter 3

Knowledge in the Software Engineering Process

In this chapter, we describe our perception of the different kinds of user, different kinds of knowledge types and representations, and different kinds of knowledge sources. We are preparing the reader to better understand our approach to solving the software knowledge-related problems discussed in chapter 1. We will describe the knowledge needed by the various people involved in software development, the pros and cons of the different kinds of knowledge representation, and the kinds of knowledge sources, noting their advantages and shortcomings.

3.1 Kinds of Software User

The development and use of a software system involves a number of people; we can divide them into two categories: customers (often termed "users") and systems personnel (who are themselves users of software development tools).

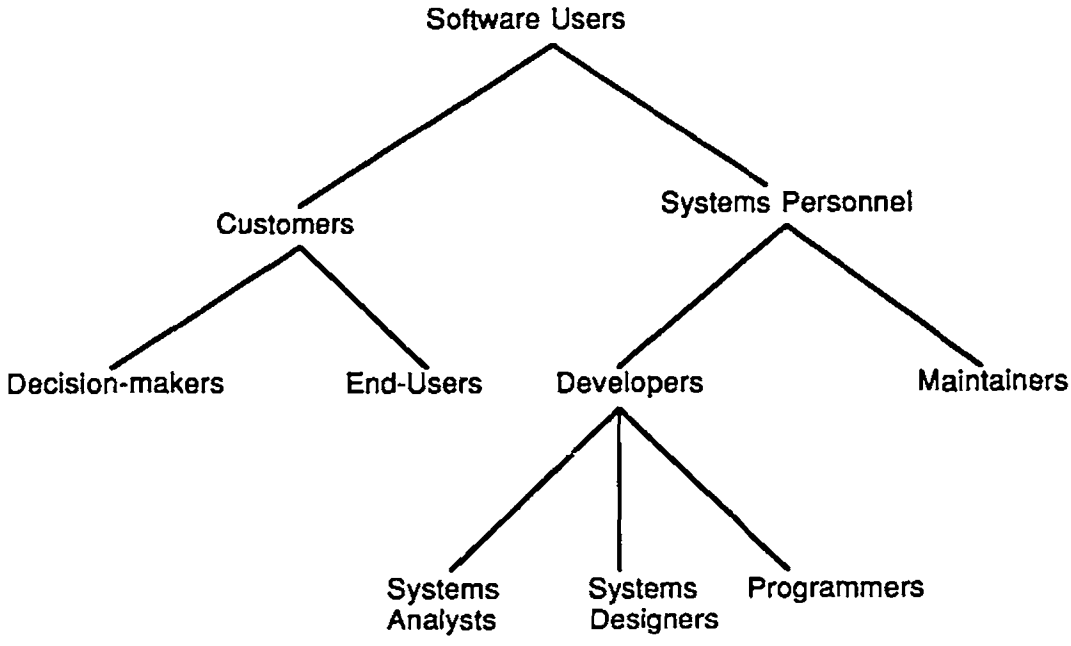

Fig 3.1 Kinds of Software Users in the Software Engineering Process

Customers include decision makers and end-users. Decision makers, such as bank managers, can be defined as those people who choose to integrate some of the work in their institutions or organizations with a computerized system. The decision makers often specify high level requirements for the new system; in addition, their primary concern is generally that the system satisfy their needs and be easy to use. End-users are those people that will use the developed system and they, too, have specific requirements. Primarily, end-users look for a system that includes a user-friendly interface. Decision makers can either be end-users or they can be participants in the agreement between end-users and developers. Customers communicate their requirements and the domain knowledge to system developers by using documents and through discussions. In general, customers must interact with the developers at the requirements level and must be shielded from the complexity of specifications and system design. Norman et al. [Norman et al. 92] explain the role of the customers in the ultimate quality of the product: "The quality of upstream products is determined by how

well systems personnel can get users and managers involved in development".

Systems personnel can be divided into two categories: developers and maintainers. Developers include: systems analysts, systems designers, and programmers. Systems analysts are responsible for specifying requirements and for describing the application domain so that systems designers can understand it and map its concepts into specifications and system design; evidently, the systems analyst must understand the requirements and the application domain before describing them. Then, systems designers design a system that programmers can implement using a suitable programming language. The systems designer must clearly understand the analysis of the requirements and the application domain, and must be capable of differentiating between relevant and irrelevant domain concepts. For the programmer, it is very important to understand the system design and its rationale. Closely related to the developers are the system maintainers. Maintainers either fix bugs in the developed system or modify/extend an existing system. Their primarily role is to understand the system and how it should be maintained. For both the developer and the maintainer, knowledge is most valuable when it is well-represented and complete. Thus, knowledge management plays an important role in the development and maintenance of a robust software system that maximizes the productivity/cost ratio.

3.2 Kinds of Knowledge Representation

Knowledge can be represented in many forms; e.g. natural language, mathematical expressions, diagrams, tables, and actual-code or pseudo-code.

Knowledge expressed in *natural language* is usually written in documents in the form of statements. Natural language reflects the manner in which humans communicate and is easier to use than other

forms of knowledge representation for most purposes. However, if a computer is to be actively involved in communication, there must be at least some level of formality, such as syntax rules and a parser to verify that formal rules are followed when analysing statements expressed in natural language. Natural language causes problems due to the ambiguity of grammar rules and semantic interpretation rules.

Knowledge can be represented in very *formal notations*, such as mathematics. Although this kind of representation is very reliable and precise in expressing knowledge, since it has a logical foundation, it has some shortcomings. It is very difficult for many people to familiarize themselves with and to use mathematical notations. Everyone involved must be familiar with these notations. In addition, many concepts or relations cannot be described mathematically.

Knowledge expressed in *diagrams* can communicate knowledge very effectively. These diagrams often involve the use of notations, conceived by a variety of people, and using varying degrees of formality. Diagrams usually consist of linked nodes. Knowledge is represented both inside nodes or on the links between nodes. Although diagrams may contain complex notations, they can be very expressive in manipulating and in communicating knowledge. This kind of representation is often easier to understand than purely textual representation.

Knowledge expressed in *tables* is also very useful for showing many aspects and features of concepts. Knowledge is represented in the form of rows and columns. It can show common properties and values among these concepts. It can also clarify the difference between two or more concepts. Spreadsheets, like Lotus 123 and QuattroPro, have become very popular over the past few years due to their tabular nature and the functional dependency among their rows and columns.

Knowledge expressed in *actual code* is often difficult to follow; it often requires a great deal of effort to understand the

implementation rationale. Often, maintainers spend hours trying to understand the existing code before attempting to perform any changes in the system. Even so, a programmer may get lost after working for some time in the same system or even within the same module. Knowledge about code itself is often inadequately documented, usually only as unstructured comments.

Another kind of knowledge representation is *pseudo-code*. This representation takes the middle ground between design and implementation; it must provide a smooth transition between them. However, it is often not associated with the system once it has been developed, nor is it frequently updated.

3.3 Kinds of Knowledge Sources

Knowledge can be captured and extracted from many sources; e.g. software systems, documentation, and human experts. These differing sources have both advantages and disadvantages to people seeking knowledge. We discuss next some such sources.

3.3.1 Software-Based Systems

Software-based systems should provide the most reliable source of knowledge. Such systems can be divided into programming language environments, CASE tools, knowledge-based systems, and hypertext systems, as discussed above.

Programming language environments provide the developers with tools to help them find knowledge related to the implementation phase, to access existing libraries, and to find existing code. Programming language environments allow developers to store comments everywhere in the system, but usually have no facilities for searching these comments.

CASE tools provide the developers with diagrams containing knowledge about the system being developed. Some CASE tools allow developers to generate documented diagrams.

Knowledge-based systems provide the developers with a knowledge base in which to access knowledge about previously-developed or existing systems. These knowledge-based systems have some knowledge representation capabilities which allow them to store, represent, and retrieve knowledge.

Hypertext-based systems provide the developer with capabilities to manage knowledge through the use of nodes and links. Knowledge is stored in nodes which may contain other nodes, and knowledge between nodes explains the relationships between the different nodes.

3.3.2 Documentation

Conventional documentation is probably the major source of knowledge about software. Documented knowledge is mainly expressed in natural language, augmented by diagrams, mathematical expressions, or tables. Primarily, documents are intended to provide the developer with a clear picture of the system developed. A typical document must be conceptually and physically organized, consistent and correct, contain all necessary knowledge, and be easy to access and to manage. Documentation exists in different media; e.g. paper, electronic files. Electronic documentation is more efficient than documentation on paper, as the former can be more easily browsed, manipulated, and transferred than the latter. Some documents are structured, easy to read, and organized while others do not follow any format and are major sources of misconception and error. Some documents are written as concisely and precisely as possible, while others poorly represent the actual system, are far from accurate, and do not reflect the system's main aspects. Some documents fall out of date while others are kept up to date. Some documents are written

by people that do not have sufficient knowledge about the system while others are written by professionals. Sametinger et al. [Sametinger et al. 92] explain the importance of documentation in the maintenance phase, the most time-consuming phase in software development: "Development programmers hate producing documentation which is, therefore, almost never consistent or complete. Maintenance programmers need documentation to understand the software system for which they are responsible."

Documents about software systems must be kept up to date as the systems evolve and must reflect the current system. These documents should contain a variety of representations of the software system. They must also be accurate, organized, readable, and easy to maintain and access. Understanding programs is one of the most time-consuming activities in software maintenance. By improving the availability of complete and up-to-date documentation, we can reduce software costs.

3.3.3 Human Experts

Human experts are a major source of knowledge, but often much of their knowledge is inaccessible. Human experts include both domain experts and developers. They typically have a great deal of knowledge, and some may have made notes of their knowledge (although these might be informal or incomplete). Domain experts are the most knowledgeable people within a given domain; they usually provide the analysts with the necessary domain knowledge. Developers have to act as experts in providing their expertise to other developers in order to produce efficient software systems. Often, expert developers have documents from previously developed systems. Typically, their primary method of communicating their knowledge is through verbal discussions or written documents.

3.4 Problems with Current Knowledge Sources

Often, knowledge communication presents major problems to people. Knowledge can be hard to find or may not exist; it may be scattered throughout a particular environment, it may be irrelevant or too low level or not adequately detailed, it may lack rationality, it may be inconsistent, incomplete or out of date, and it may be invalid.

Such problems are typical of current knowledge sources. Knowledge about a software system is usually distributed among source code, documentation, and experts.

We will now describe a variety of typical knowledge management problems that may occur in different tools commonly used in software development.

3.4.1 Knowledge Management Problems in Current Software Systems

Current software systems contain many knowledge management problems:

Programming language environments do not provide adequate assistance for finding knowledge; developers are often frustrated in their efforts to find appropriate library functions, procedures, modules, or classes. Often, they spend a lot of time browsing existing libraries to find what they want; but libraries are usually large, they often do not use the correct terminology, and their components are often inconsistently or incompletely described. The ability to browse existing code is limited to simple, traditional mechanisms, as knowledge is not integrated within the different parts of the system being developed. These environments do not provide any capabilities for conceiving the system from different points of view. Smalltalk-80 environment is a good example of an environment that needs better knowledge management capabilities:

- A user usually has to pass through several levels of indirection (message-sends) in order to understand a certain method.
- A programmer/maintainer must rely heavily on variable names or comments to understand existing classes and methods.
- As explained earlier in section 2.2.1, Smalltalk-80 provides only primitive tools for retrieving classes or methods.
- Indexing methods or classes depends on knowing their correct names.

CASE tools, which are intended to provide powerful assistance to developers, still lack capabilities ([Forte et al. 92] and [Norman 91]) in:

- Requirements elicitation.
- Understanding the software process.
- Support for the communication between developers and end-users and among developers/maintainers themselves.
- Knowledge representation and inferencing.
- Support for language-related problems.

Knowledge-based systems, which represent knowledge in concept hierarchies and perform some inferencing, still lack some key features:

- they have not matured enough to support an engineering approach to software development.
- (most of them) do not support natural language processing. Those who do, treat it as a front end but not part of their design.
- they cannot generate complete documentation from their knowledge base
- (most of them) lack sophisticated graphical user interface capabilities.

Hypertext-based systems, which provide an excellent medium for thinking and communication, have not yet provided satisfactory solutions for the following problems ([Conklin et al. 89] and [Lucarella 90]):

- Information retrieval process.
- Combining two important processes together: browsing and searching.
- Disorientation and cognitive overhead (as explained earlier in section 2.4).
- Brainstorming ability.
- Support for natural language-related problems.

3.4.2 Problems with Documentation

Documents that are written in human or natural language often contain ambiguity, are poorly organized, do not provide adequate assistance for finding the required knowledge, e.g. a good index and do not allow for querying, filtering, or masking, etc... to be performed on their contents. They often lack a glossary, or use terms inconsistently or ambiguously.

Problems with documentation stem mainly from the manner in which documents are produced. Documentation typically is produced by documentation teams comprised of people who were not originally involved in the project, causing a knowledge transfer problem. These people are called "Technical writers". Even if they were collaborating with the systems personnel, they often cannot produce correct and complete documentation. Developers (or maintainers) may not provide them with all the information they need or they may forget some of it. Perhaps technical writers' documentation is organized, consistent, and readable, but often, there is no standard way to ensure that their documents accurately describe the developed or maintained software system. If developers try to document developed systems, they either write the documentation in the code

itself or in a file separate from the developed environment. Often they may do this poorly, since they are not skilled writers and do not enjoy this task. Sametinger et al. [Sametinger et al. 92] point out the disadvantages of both methods. If they use separate files, there won't be any connection between the source code and the corresponding documentation file. If they write the documentation straight into the source code, the source code may become harder to follow if too many comments are embedded, and the comments cannot be adequately structured or indexed.

3.4.3 Problems with Human Experts

Human experts, used as knowledge sources, present other major problems: experts have difficulty expressing their expertise in a concrete form; when they have to communicate their thoughts to someone who is not familiar with the domain of expertise, there is often difficulty in reaching an agreement on terminology between experts and developers; they may forget some important details until after the system has evolved too far, they may skip some details assuming the developer has their same understanding; and they themselves may not be aware of some of the details in their domain. They have a good understanding of the application domain and an abstract model in their mind of how the system will be developed and how it will function. Ideally, they should make note of this knowledge, as it may help others to understand the system more quickly and easily.

Chapter 4

Conceptually-Oriented Software Engineering (COSE)

In this chapter, we describe our approach, Conceptually-Oriented Software Engineering (COSE), to developing software systems based on managing knowledge in three important *viewpoints:* domain knowledge, design knowledge, and implementation knowledge. Each will be treated within a common generic knowledge representation framework. These kinds of knowledge are:

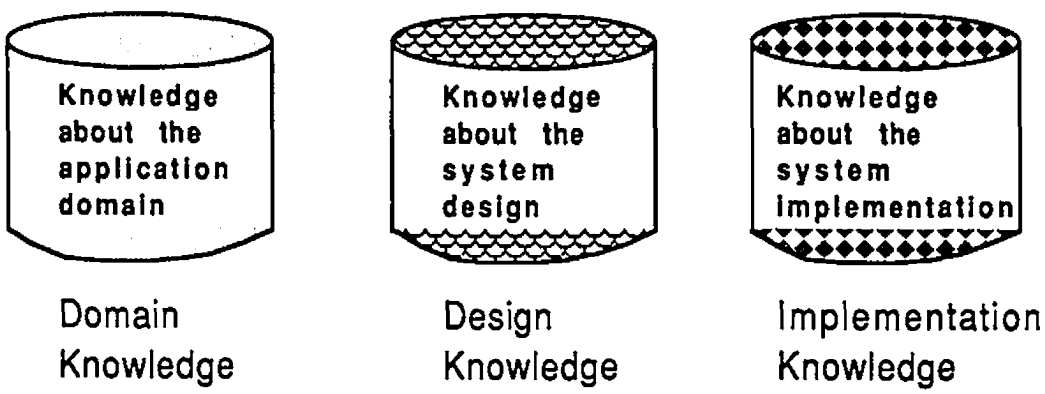

Fig 4.1 COSE: A Software Knowledge Development Approach

- *Domain knowledge:* knowledge about the application domain without any consideration of design decisions (This knowledge is mainly needed by other team analysts). We include in domain knowledge requirements statements, which can be specifically flagged as such.
- *Design knowledge:* knowledge about the system design and its rationale, usually without any dependency on the implementation language (This knowledge is mainly needed by programmers and other team designers).
- *Implementation knowledge:* knowledge about the implementation of the design including the code-level details (This knowledge is mainly needed by maintainers and programmers).

Our guiding principle is that all three viewpoints (types of knowledge) should be easily locatable, understandable, in similar format, and hence easily reusable. Knowledge maintenance can take place in any one of the mentioned viewpoints and reduces the risk of software inconsistency or invisibility. We believe that software engineering should not be separated from knowledge engineering, i.e. that software engineers need a lot of assistance with knowledge management concepts and techniques. Our approach is consistent with the view presented by [Jarke 92]: "In requirements specification or analysis, you need the freedom to define application-specific concepts and terminology. In contrast, during the design phase, you need a predefined but powerful set of constructs to represent a system perspective".

In the next three paragraphs, we will describe in more detail the three different kinds of knowledge that we believe are essential in software development:

Domain Knowledge

Since domain analysis activity involves human communication and conceptual agreement among developers and analysts, we believe that the software development process should start with a conceptual model representing all the relevant concepts of the domain. Our approach is consistent with the view advocated by [Greenspan et al. 88]: "The conceptual modelling level is necessary in order to provide a modelling platform (at a higher level than that offered by the Basic Object Level), for introducing domain-specific concepts". By Basic Object Level, Greenspan means the identification of domain objects (concepts) that will eventually be transformed into a design. The role of this model is to define these concepts, their properties and their relationships to each other to assist in assuring that the systems personnel are in close conceptual and terminological agreement with the customer. Hence our approach is to represent the domain knowledge in a conceptual (is-a) hierarchy. And since the requirements are dependent on many of the domain concepts, we represent them in the same hierarchy. This conceptual model is totally independent of any software or systems concepts or terms. Hence, it should be completely understandable by the customers, so they may validate it.

Requirements statements can be associated with each concept. Any statement can be flagged (using a CODE facet) as being a necessary, optional, or negative requirement.

Design Knowledge

Since we believe that concepts are the abstractions of objects and the majority of their properties are the abstractions of behaviour and states, we take an object-oriented approach in our design. Design knowledge is organized around a hierarchy of classes with knowledge about their behaviour and states also represented in associated hierarchies. This design knowledge is language-independent, i.e. it can be implemented in any object-oriented language, and is of possible use to anyone, except the domain expert, involved in the development process.

Implementation Knowledge

Finally, implementation knowledge is represented similarly in another hierarchy that captures the details of the system and is therefore dependent on the language used for the coding. This knowledge represents all the details of the implementation, which for object-oriented programming, are mainly descriptions of the classes and methods of the system.

Following our three viewpoint approach (COSE), our environment will assist the following types of users:

- Domain experts
- Analysts
- Designers
- Programmers
- Maintainers

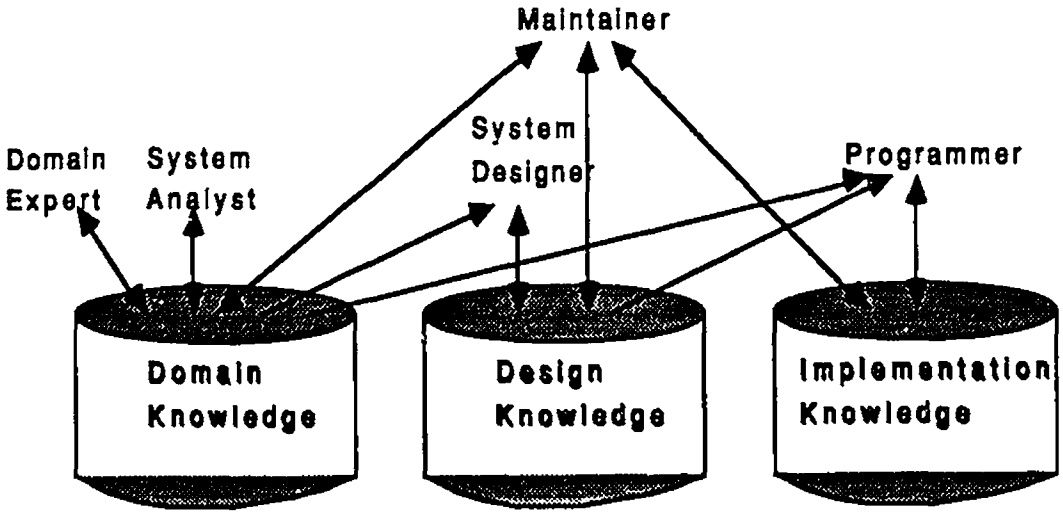

Fig 4.2 Users' Benefits from COSEE

Kozaczynski [Kozaczynski 91] explains: "An effective way to support software understanding is to answer different kinds of questions that the user may have".

For each of these three types of knowledge, we have another orthogonal dimension in which we differentiate between "generic" and "application-specific" concepts; i.e. concepts that are more general than this application and concepts that are specific to the application. These differences will be described in more detail in section 5.2.1.1 when we describe the ATM example.

Before we describe the viewpoints in detail, we explain several basic concepts behind our representation. Knowledge is divided into units we call *concepts*. A concept is anything we want to say something about, often denoted by a noun phrase. To express the properties of these concepts, we make what we call *statements* about each concept. The statements themselves can be arranged in a hierarchy sometimes called the property or the statement hierarchy.

We also introduce the notion of the formality spectrum: Knowledge representations can have three "degrees" of *formality* :

- *Informal Representations* are mainly used in the domain knowledge to capture the relevant concepts. They are unrestricted and do not have any syntax or semantics. Currently most knowledge-based systems contain some kind of informality often through comments.

- *Semi-Formal Representations* take the middle ground and have some parts of the representation interpretable by the computer, but others only for human users. For example, task specific objects (e.g. decision, goal, claim) can be semi-formal objects with their own attributes and formally related, but the system may allow these attributes values to be filled in by designers in form of informal descriptions or other semi-formal objects that the user might choose to create. The system then processes the descriptions to the extent that they have been formalized, but leaves others for human processing. The appeal of the semi-formal representation approach is that there is relatively less overhead in capture (in fact, semi-formal representations can be easier to deal with than informal representations by suggesting what information is expected and defaults), yet users can define computational operations exploiting the formalized part of the representation. [Conklin et al. 89] is a good example of a system that uses a semi-formal representation. The decision of choosing the degree of formality should be dependent on what the user needs: e.g. an automated assistance is required or not. The more formality the more orientation towards the automation process and machine assistance.

- *Formal Representations* have the advantage of having formal semantics and being interpretable by computers and having well-established inference procedures, but they may be hard to create and comprehend. Mathematical notations can be attached to the knowledge and the machine may be able to handle them properly. For example a machine can do serious theorem proving using

first-order logic, but VDM [Jones 89] cannot be executed because it embeds too much mathematics and logic. Also, the domain knowledge needed to understand formal representations is often missing. On the other hand, informal representations are easy to create and natural, but they are not interpretable by computers and rely on human processing alone.

In the next three sections, we will explain our approach to software development, COSE, by describing the problems resulting from poorly managing each of the three types of software knowledge discussed earlier, how other approaches try to solve these problems, and how COSE proposes to deal with these problems.

4.1 Domain Knowledge in COSE

Requirements analysis has been recognized as one of the most critical and difficult tasks in the construction of software systems [Reubenstein et al. 91]. As one moves from an informal description of an application to a formal (or at least semi-formal) representation of it, errors are often introduced due to incorrect understanding of the desired properties of the system. Reubenstein et al. seek to solve this problem in the Requirements Apprentice project: "The focus of the RA is on the formalization phase that bridges the gap between an informal and formal specification. This is a crucial area of weakness in the current state of the art". In the preliminary phase of software development, the systems analyst must have some knowledge about the domain in order to produce an accurate description of the application domain. If the analyst has none initially, there must somehow exist a way to acquire it; either by consulting experts or appropriate materials. Often, communication problems and misunderstandings occur between the customer and the analyst. Differences of terminology and concepts often create problems in the analysis process. The problems of misinterpretation and bad communication also arise when groups of analysts cooperate in

analyzing requirements of complex problems. In such situations, different analysts usually focus on different parts of the problem. In doing so, they may develop different ways to referring to domain concepts, or make different assumptions about them. They may choose to model the domain in different ways, and make different simplifying assumptions. As a result, the evolving system description is sometimes incomplete and does not precisely reflect the application domain.

Current software engineering analysis methodologies and tools (like structured analysis and CASE tools) do not provide sufficiently comprehensive means of representing, defining, and managing the concepts of the application domain. These tools represent only certain kinds of knowledge and leave the rest out. Existing object-oriented analysis and design methodologies (e.g. [Rumbaugh et al. 91] and [Booch 91]) are built around the computer notion of "object", ignoring the natural description of the domain concepts and the freedom to express one's conception about the domain. These tools and methodologies are very good only for a certain type of knowledge representations (e.g. finite state diagram, entity-relationship diagram) while others have to be dealt with purely informally or unstructured English comments. Even if we use these tools in our development, the number of bytes of knowledge they end up storing can be considerably less than the number of bytes that are stored as natural language documentation that has to go on with it. Evidently, there is much more knowledge that these tools cannot capture, i.e. it still has to be captured in a natural language form. Our approach seeks to eliminate these limitations and constraints in describing the domain. Our approach, can be viewed as an extension of the object-oriented analysis approach since we capture a wider variety of domain knowledge types not just knowledge pertinent to the object-oriented design task.

We believe that a better solution for these analysis problems lies in conceptual analysis (CA). By conceptual analysis, we mean a description of all the different concepts in the domain (including

requirements analysis concepts) and the relationships between them. Our approach to analysis is to represent these domain concepts by both a conceptual hierarchy and a hierarchical description of properties as well. We capture both domain knowledge and requirements in a unified framework that describes both the domain and the corresponding design and implementation concepts. Explicit pointers will define these correspondences. For example, in the ATM example, the domain concept of a bank account may be reflected in a design object class also called BankAccount.

This domain knowledge is independent of any software perspective view. All concepts introduced and described must be validated in principle by the domain expert. Software designers will rely on this kind of knowledge before and during the design phase. Others (e.g. implementors and maintainers) will rely on it after. Our approach in capturing the domain knowledge can be used for any kind of design, not only object-oriented design on which we are going to focus.

4.2 Design Knowledge in COSE

Software design is perhaps the central activity in software development; errors at this stage are costly to rectify, and the quality of a design greatly affects the flexibility and adaptability of the final system. Design is often regarded as an art performed by designers who start their work by trying to understand the described domain and by mapping requirements and specifications into a complete software design system. The design of software is a complex process requiring the software designer to simultaneously perform a variety of knowledge-intensive activities. These include the exploration and analysis of design alternatives, the consideration and reuse of existing components and solutions, the learning of the management of design goals, dependencies, and partial solutions, and the recording of design decisions. The system design phase determines how the system performs the functionalities that are required.

Various design methodologies have been introduced (e.g. [Rumbaugh et al. 91] and [Wirfs-Brock et al. 90]). Each attempts to give the designer a conceptual framework or *ontology* to assist in structuring the design. For example, Wirfs-Brock et al. [Wirfs-Brock et al. 90] introduce some design concepts that help structure a design. They define the concept "Responsibility" as the service that an object can provide, the concept "Contract" as the set of all requests a class (a client) can make from another class (server), the concept "Collaborations" as the set of requests a class can make of other class in order to fulfill a certain responsibility, and the concept "Protocol" as the set of cohesive responsibilities provided by a class. In our design knowledge, class responsibilities can be described and collaborated classes can be specified for every responsibility. Mainly, a programmer needs to understand why a specific responsibility is introduced and how a class interacts with other classes. Following [Wirfs-Brock et al. 90] methodology, we also partition the class behaviour into protocols to reduce the design complexity.

Although our approach could be used in any kind of design, our effort is focused on the design of object-oriented systems. We believe that object-orientation provides a means to associate software components to entities of the application world, thus making the design more natural. The object-oriented paradigm is one of many ways to achieve modularity in a program. In an object-oriented design approach, the result of a system design is a group of classes with their methods. Our approach to software design is to capture all knowledge about the design in a framework that provides a clear and explicit representation of class hierarchies with inheritance of key properties such as responsibilities (behaviour) and attributes (state attributes, fixed attributes, and changeable attributes). A programmer or maintainer needs to know the structure of classes, their relations to each other and why they are introduced in the design. Eventually, at implementation time, more detailed knowledge about the methods and the states (variables) of objects will need to be recorded. But a design is an abstract representation of a set of objects being created and all their properties; some implementation

details are not expressed. Therefore, a design should be viewed as a repository of knowledge about objects. The design process can be viewed as an evolution of object descriptions such as adding more information and backtracking and exploring alternatives. Design knowledge should include not only the design classes and responsibilities but also the design rationale, i.e., why certain choices were made.

[Ramesh et al. 92] explain the importance of capturing the design rationale: "Current practices for describing designs emphasize the representation of outputs or artifacts that result from this process and ignore the rationale behind their creation. There is growing recognition that capturing and representing such process oriented aspects of systems design will increase the productivity in development and maintenance of systems". Often designers introduce classes and methods without documenting the reason of choice or the purpose of creating them and how they must interact with each other.

An important issue for the documentation of design knowledge is the notion of a formality spectrum [Lethbridge 91]. Formal methods provide a means for documenting design knowledge in a formally verifiable manner supportive of design reuse. But many users cannot understand formalisms.

The approach we have taken is a semi-formal approach in our design knowledge capture with a range of freedom left to the user to increase or decrease formality between the two boundaries: Formal and informal representations. We try to provide a framework where any formal description can be intimately linked and correlated with any kind of less formal description (e.g. natural language, ClearTalk, or finite state diagrams).

4.3 Implementation Knowledge in COSE

In this phase, a realization of the designed system must be achieved as executable code. Often poor implementation can create a major problem in the maintenance phase. A program sometimes is non-readable, non-documented or poorly documented. A maintainer may spend a lot of time trying to understand the implemented code or why some code had been patched in a specific place. Even a developer can experience difficulties trying to remember or understand the code written by himself sometime before. Flowcharts and pseudo-code are of limited use especially with large developed systems. Structured programming has been a forward step towards program readability and comprehension; however, many believe the object-oriented programming represents an advance over the traditional software processing.

An object-oriented developer/maintainer needs to get answers to many kinds of questions:

- What is the purpose of a specific method?
- Who is the implementor of a certain method?
- What are the different messages sent from within a method?
- What are the types of the arguments of a method pattern?
- What is the expression returned by a method?
- What are the different methods that have similar names and purposes and how do they differ from each other?
- What are the collaborative classes for a certain method?
- What are the unusual properties that can help them during the implementation?
- What is the history of the designed method itself?

We believe that it is necessary to capture all these kinds of knowledge in an effective software development environment. This knowledge should be linked, where appropriate, to the design, the domain knowledge, and the programming environment as well. By consulting this knowledge, a maintainer will be able to get a more reliable and more accurate information than if consulting the development environment or other software knowledge sources including the original developers.

Our approach keeps this kind of knowledge constantly accessible during all stages of the software development process. We emphasize the importance of such knowledge as an interactive assistant to the developer or the maintainer.

Implementation knowledge is a language-dependent, i.e. it differs from one language to another. In our case, since we selected Smalltalk-80 as our implementation language, our implementation knowledge is specifically about Smalltalk-80 constructs (classes and methods).

Chapter 5

Software Development Using a Knowledge Management System

In this chapter, the main part of the thesis, we will explain our prototype environment COSEE, followed by an analysis of the example that we have worked on: The ATM (Automated Teller Machine).

Our approach to confronting the software crisis is to use a unified knowledge management system to attempt to capture all knowledge involved in software development with links to the actual programming environment. We believe that the software development environment should include a repository of (ideally) all knowledge needed by both developers and maintainers. Our environment is intended to encompass the entire life cycle of a software system from the gathering of requirements, and formulation of specifications to the maintenance of the resulting code.

Besides being linked to the programming environment (Smalltalk-80), our environment could be linked to other subsystems and tools:

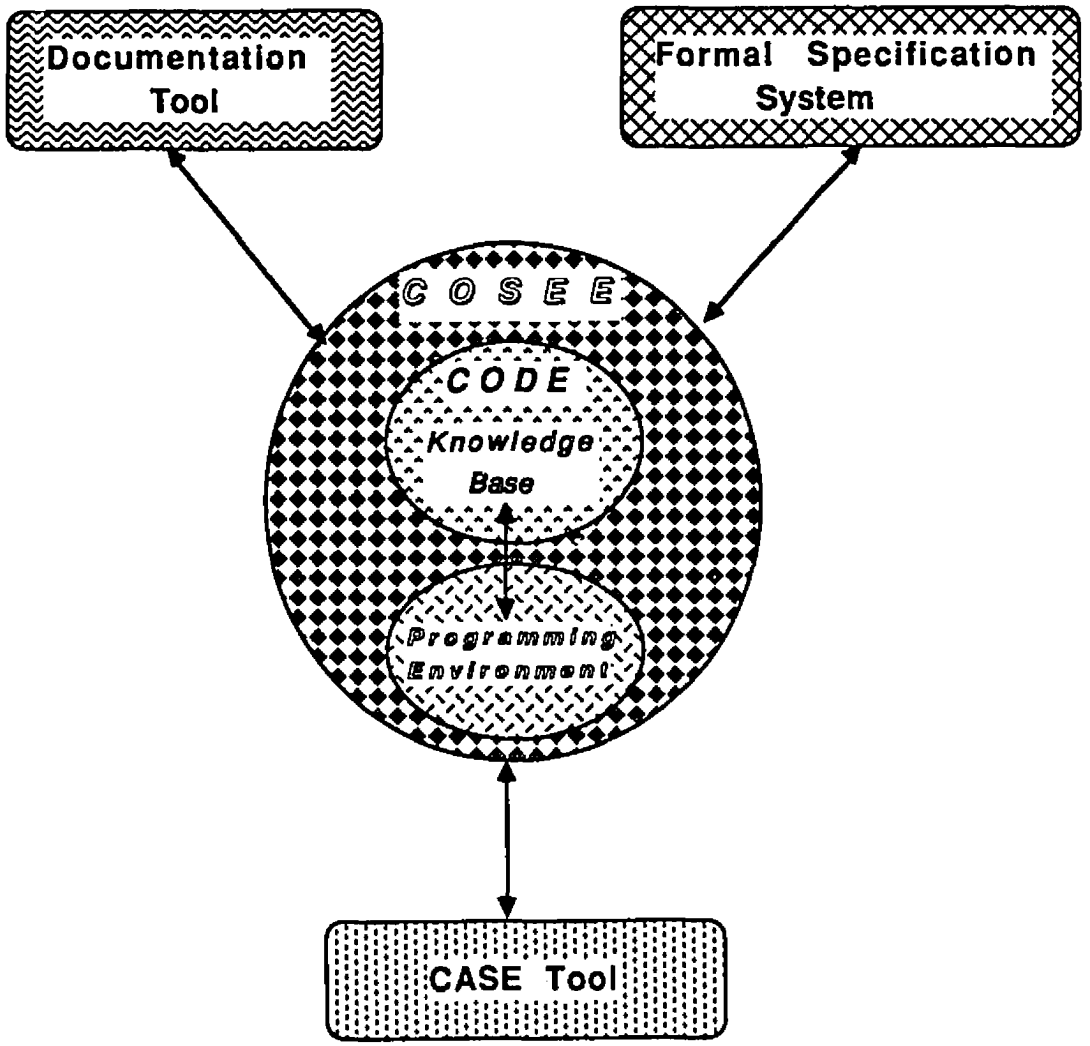

Fig 5.1 COSEE Linked to Other Systems & Tools

- A CASE tool (e.g. such as ObjecTime [Selic et al. 92]) to benefit from its software engineering capabilities as explained earlier in section 2.2.
- Formal specification systems (such as VDM [Jones 89] or [Boudriga et al. 92]) to verify design-level system components.
- A documentation tool (e.g. Framemaker) to have a complete conventional documentation of the system.

Our model goes beyond the Waterfall Model which treats the development of software as a linear process consisting of a series of phases. In particular, the Waterfall Model fails to sufficiently take into account the knowledge-intensive activities of requirements and design analysis. As a result, the final implementation is often significantly different from the original the requirement specifications, and the knowledge is often distributed in various places in incompatible, uncoordinated, or inconsistent formats. Hence, maintainers have trouble in accessing the knowledge that might be outdated, unorganized, or incomplete. Another major problem with this model is the lack of *design rationale* that is rarely or poorly captured during software development. Design rationale is currently captured in a separate subsystem from the design documentation subsystem [Ramesh et al. 92]. By allowing designers to integrate design rationale into the knowledge base in forms of statements, design knowledge can be easily understandable and reusable.

The main features of our environment are* :

- It provides a unified medium of interaction for the development process and assists the user who can therefore better maintain the semantic consistency of the software system as it evolves from its specification to its implementation.
- It provides a common environment for communication among different subsystems that have difficulties in providing an integrated and consistent knowledge.
- It provides the same functionalities of some of existing non-integrated subsystems (e.g. the design rationale as explained above).
- It helps clarify natural language descriptions and specifications.
- It helps domain experts and systems personnel reach agreements on terminological problems involved in the developed system.

---

* We will elaborate these points below

5.1 COSEE: Conceptually-Oriented Software Engineering Environment

COSEE stands for Conceptually-Oriented Software Engineering Environment. The basic idea is to extend the object-oriented model; to extend the notion of object to something more general that we call "Concept" (a description of all concept properties, not just behaviour and states), hence the name Conceptually-Oriented. COSEE is our prototype of such a system: a software development environment built on top of the knowledge management system CODE4 (the current version of CODE). It is also linked to a programming environment; in our case the Smalltalk-80 environment. In COSEE, we represent our three different viewpoints of knowledge involved in software development: domain knowledge, design knowledge, and implementation knowledge. These three kinds of knowledge are stored in a single knowledge base but can be isolated. Throughout this section, we will refer to the example we will use in the following section (the ATM example inspired from [Wirfs-Brock et al. 90] and [Rumbaugh et al. 91]).

Pointers among these viewpoints allow the knowledge base to contribute to a unified and integrated software development environment. The user (e.g. a software developer or maintainer) can easily browse among viewpoints. For example, if the user is in the implementation domain studying an object called BankAccount, the user can go all the way back to the domain knowledge where he/she can learn more about the concept 'bank account' from the banker's point of view; this helps him/her to understand the object he/she is working on and possibly recognize that there is a problem (e.g. the banker said there are three kinds of account but only two seem to have been implemented).

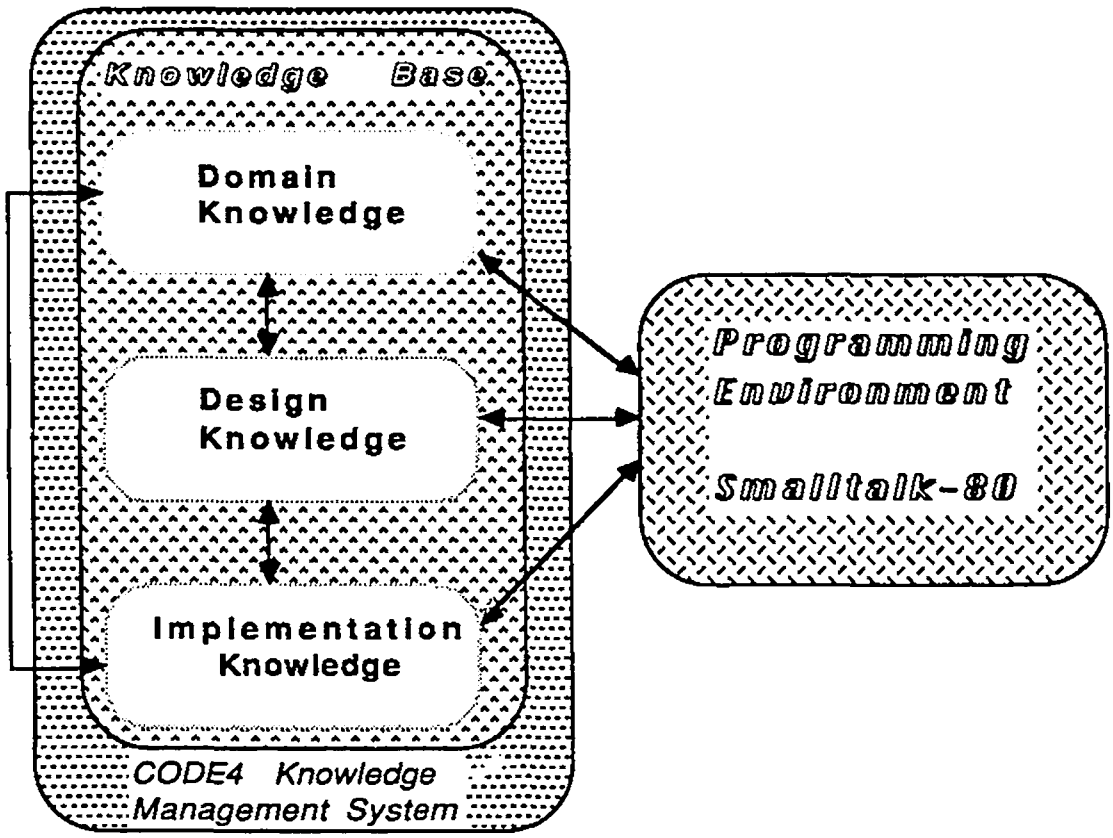

Fig 5.2 COSEE: three viewpoints linked to the programming environment

Besides linking these three kinds of knowledge, we provide direct links to the actual programming environment (Smalltalk-80) so that each type of concept can be closely linked to the implementation itself. This essentially replaces and augments what is normally found only as code comments or other documentation written separately about the actual code itself. There are pointers that allow the user to jump directly from the knowledge base to the Smalltalk-80 browser in both directions, allowing the user to understand how a concept and its properties in the domain knowledge, a class and its behaviour in the design knowledge, or an object and its methods are mapped into their corresponding Smalltalk-80 classes and methods.

Based on the assumption that the conceptually/object oriented development approach will allow the computer to be used as an

intelligent assistant in solving complex problems, we have developed an environment that assists software developers to manage their knowledge in a manner more naturally suited to the way they think and communicate.

Our approach is similar to the Programmer's Apprentice in representing three kinds of software knowledge: domain (requirements) knowledge, design knowledge, and implementation knowledge. It can also be considered as the result of combining LaSSIE (conceptual and architectural knowledge) and CODE-BASE (implementation knowledge). The overlap is strongest in the recognition of the need for multiple perspectives in the knowledge representation and multiple views to support the users' interactions with the knowledge.

5.1.1 CODE Basic Concepts

In this section, we briefly introduce the basic concepts of CODE4 [Skuce et al. 92] to the reader who is not familiar with the system. We will leave the knowledge representation features and the user interface features to section 5.3

Concepts, Predicates, and Statements

Everything one might desire to discuss and hence represent is termed a "thing". A concept for a thing X is a set of statements about X, plus any statements about the statements. For example, there is the concept "bank account"; i.e. all the knowledge CODE4 has about bank accounts. These concepts are arranged in an is-a hierarchy in which more specific concepts inherit properties from more general ones. A concept with all its properties is described by a "Conceptual Descriptor". A conceptual descriptor is like a frame; it is a data structure consisting of a variable number of slots or statements that consist of facets (smaller statements representing incremental addition to a statement) and their values. Statements of a certain

concept correspond to slots of a frame. Often one wants to create a conceptual descriptor describing a property. CODE4 has facilities for linking from a property of a concept (i.e. a single statement about it) to another concept description which gives further details.

Things have properties that are referred to by predicates, i.e. the notion of property is very general and is intimately associated with those of a predicate and a statement: If one can make a statement about a thing X, then this statement is said to express a property of X, and for X have a property P, one must be able to express P by some statement about X that uses an appropriate predicate. For example, 'balance' is a property of a bank account and 'has-balance' would be a predicate that refers to it.

Statements have two essential parts: The subject, which refers to a thing, and the predicate which refers to a property of that thing. A predicate is an abstraction of a statement: it does not make a statement, but can be thought of as a template or basis for possible statements that can be made by adding suitable information, at least a subject. A concept for X is described by all the statements having X as subject, plus their component statements, etc. A concept (descriptor) can be as small as a single statement. For example, if we say "ATM has the purpose of performing financial services", then 'ATM' is the subject, 'has purpose' is the predicate of the statement, 'purpose' is a property of an 'ATM', and 'performing financial services' is the value of the predicate. Statements can have a range of formality ranging from very informal (where there is no intended translation into more formal representation) to extremely formal (for example, it might be translatable directly into a first-order logic, Prolog or some other formal language). One of the deliberate design goals in CODE4 is to provide this flexibility where one statement might be informal (like a comment about something) and another statement might be formal (like logical constraint expressed in formal language). Thus, a CODE4 user can encode knowledge with varying degree of formality. The more the degree of formality, the more inferences can be done automatically. Like in most, if not all,

knowledge representation systems, CODE4 can make hierarchical conceptual descriptions based on the notion of inheriting properties.

Hierarchical Structures

Two hierarchical structures are central to CODE4: The is-a hierarchy (concept hierarchy) which represents the abstraction relationships between concepts, and the predicate hierarchy which allows the arrangement of predicates and hence statements hierarchically. We explain these further:

> The subjects of all statements, and hence all concepts, are located in an inheritance (or "is-a") hierarchy or *"Concept Hierarchy"* that permits multiple inheritance of statements. The purpose of this hierarchy is conventional: to permit taxonomic structuring of knowledge and property inheritance. A (subject) node referring to a thing may be created in the hierarchy without actually making any statements about it, except to identify its parents in the hierarchy. But as soon we make statements about a thing, these form its concept descriptor.
>
> The other hierarchical structure is termed the *"Predicate Hierarchy"*. All predicates are arranged in a separate hierarchy in which the partial order is interpreted as "implies". The "top" of the predicate hierarchy is a predicate meaning "has a property". Each predicate is represented as a CODE4 object (a thing) which is an instance of the primitive concept 'predicate', from which it inherits properties that predicates have. We can also refer to this hierarchy as an "implication hierarchy" meaning that if P2 is a subproperty of P1, then the statement of P2 about a subject S (a concept) implies the statement of P1 about S. The *"Statement Hierarchy"* is the direct result of the predicate hierarchy. For example, if we

say that "making a deposit" is a subproperty of "an action on ATM", then the statement of "making a deposit" about ATM implies the statement of "action on ATM" about ATM, i.e. there are actions on ATM.

The concept hierarchy, predicate hierarchy, or statement hierarchy can be displayed in outline or graphical nodes (see, e.g., Fig 5.7 or Fig 5.8). The outline hierarchy displays concepts, properties, or statements with indentation to show the hierarchical relationships among them (children or sibling relationships). The graphical hierarchy displays the concepts, properties, or statements as nodes with links among them highlighting their relationships.

5.1.2 Representing Knowledge in COSEE

In this section, we discuss how we use COSEE to represent the different kinds of knowledge needed for software development: domain knowledge, design knowledge, and implementation knowledge.

5.1.2.1 Domain Knowledge

Domain knowledge is stored in CODE4 conceptual descriptors that contain knowledge in the form of statements about the domain concepts. All concepts deemed to be important in the domain are described by the domain expert or the systems analyst. Often they are first described in an informal way, but COSEE can make this knowledge much more precise.

Concepts in the domain knowledge base are not restricted to potential candidate object-oriented classes but rather they can include any concept regardless of whether or not it is relevant to the system design. It will usually not be known until later which concepts will become design object classes. The person building the

knowledge base has to decide how much knowledge to encode. Knowledge might be irrelevant to the current task but may be useful to another subsequent task and often in the beginning of a project, it is not possible to tell which knowledge will be needed; so the systems analyst may collect a lot more knowledge than is needed. These concepts, once captured in the knowledge base, must be accepted by the domain expert. Systems analysts and domain experts must work together towards an agreement on a unified knowledge base; CODE4 serves as a medium of communication between them. Other knowledge that is relevant to the development process itself could also be captured in the knowledge base; e.g. historical knowledge (like the names of the people involved in the analysis, time, location,....) and general comments.

Concept properties (i.e. statements about concepts) include anything a domain expert wants to say about a concept and are not restricted to any particular kind such as the actions performed by the concept or its attributes. Major kinds of properties include:

- **Purpose:** The purpose of the thing itself.
- **Related Things:** The things related to the thing being described.
- **Parts:** The different parts that compose the thing.
- **States:** The different states that the thing can be in.

Later on, we will illustrate more domain concept properties when we discuss the ATM example.

These properties can be very general or very specific. They can be very general to help other domain experts agree or disagree on their correctness, to help other team developers understand these concepts without any ambiguity, to reduce the time maintainers spend to understand the system, and to remove the possibility of errors among people involved in the software development in general. Or they can be very specific to specify some details that are essential for the developers to design a reliable and accurate system.

Properties can be treated as concepts, i.e. a domain expert can describe these properties in more detail. This can be done when it is felt that a property is not sufficiently clear, i.e. to represent additional information that we want to associate with a property in general or a particular statement.

Capturing all the domain concepts in a knowledge base ensures that the systems designer will have more freedom and more knowledge in designing a reliable and a flexible system that can be easily extended in the future.

After the domain knowledge has been captured in the knowledge base, the next step is the identification of potential objects for the design phase, using any Object-Oriented analysis technique (we do not focus on any particular technique). Flags attached to concepts that are considered as potential objects can help differentiate between those from other domain concepts. This step can be a product of the collaboration of systems analyst and systems designer.

5.1.2.2 Design Knowledge

By design knowledge, we mean general systems design knowledge without any notion of implementation details. This knowledge could be reused for other designs. Since our approach is an abstraction of the object-oriented approach, we will assume that the design will be object-oriented design. But we wish to emphasize that any design methodology could be used; not only object-oriented. This design knowledge will specify classes of objects together with their properties such as attributes, behaviours, and states. Design objects are represented in an is-a hierarchy with the Object class at the top as usual.

Pointers in the domain knowledge help clarify the mappings from domain concepts to design objects. A pointer is attached to

every domain concept which has a corresponding object in the design knowledge, and vice versa.

Not only will object structure, responsibilities (behaviour) and attributes be recorded in the design knowledge base but also other types of design knowledge such as design rationale can be recorded. The basic idea is to record answers to questions that new designers, programmers and maintainers will frequently ask, such as:

- What is the purpose of creating a specific class?
- Why has a certain concept been treated as a class and not some other way?
- What are the different services provided by a class and what are those inherited by that class?
- What is the difference between two (or more) methods with the same name but implemented by different classes?
- Why has a service provided by a system been treated as a class and not as a behaviour in the design?
- What are the composite classes that compose a certain class?
- Why is a class designed as a subclass of another class even though it is not conceptually correct?
- How and why have decisions regarding a specific design been made?
- How do different classes communicate and what are the different messages sent from one class to other classes in order to implement a certain method?
- What are the different states of a certain class and how are they represented in the system?

Besides the above mentioned knowledge, system personnel may need to know other related technical information (class category, class comments, ...). Other historical knowledge (the designer names, the

time, ...) and any general comments could be recorded in the knowledge base.

Thus, design knowledge mainly contains knowledge about classes, their structures, and their behaviour that is divided into responsibilities and collaborations, their states, design rationale statements, and any other relevant knowledge necessary to the systems designer, the programmer, or the maintainer.

5.1.2.3 Implementation Knowledge

Implementation knowledge is knowledge about the implementation of the software; i.e. the actual code. It is mainly intended to capture details of a particular implementation to assist in a consistent and accurate programming, which in turn assists in system maintenance.

Although we used the Smalltalk-80 language as our implementation language, our object-oriented design could be implemented in any other object-oriented language. Knowledge about Smalltalk-80 classes and methods is derived from the design captured in the knowledge base. This knowledge includes knowledge about the message pattern, the type of arguments, the value returned when a message is sent, the different messages sent and their receivers from within a method, method comments and purposes, the instance and class variables, and the instance and class methods. The goal is to provide most of the knowledge the programmer normally looks for in the Smalltalk-80 browser directly within COSEE. The only thing the programmer would need to go to the Smalltalk-80 browser for is to study some details of the actual code or to edit it. All other knowledge should be available in COSEE.

Pointers exist between a class (or a method) in the design knowledge and a class (or a method) in the implementation knowledge to help clarify the link between a designed and an

implemented class (or a method). These pointers show how classes (or methods) are implemented once they have been designed. By following the links between the viewpoints, a programmer can easily trace the ideas behind a piece of code all the way back to the domain knowledge if needed.

Further pointers link the implementation knowledge base in CODE4 and the Smalltalk-80 browser to assist in ensuring that the implemented code matches the implementation knowledge and to help find actual code. This link provides an integrated view between the knowledge about the developed software and the development environment (this will be illustrated in section 5.2). Also, COSEE can assist programmers in reverse engineering as follows: The programmer can select a class concept in the implementation knowledge and ask COSEE to automatically generate, in the same implementation knowledge, a subhierarchy of subclasses corresponding to one in the Smalltalk-80. COSEE will find the Smalltalk-80 class that corresponds to that concept, if it exists, and create a subhierarchy consisting of its subclasses with their methods and variables (instance and class variables). The programmer can also ask COSEE to generate only one of the subclasses of the Smalltalk-80 class corresponding to this concept. This is done by allowing the user to select a subclass from a pop up window consisting of all the concept subclasses, if there exists a Smalltalk-80 class (with subclasses) corresponding to the selected concept.

5.2 The ATM Example

In this section, we illustrate our approach using the ATM (Automated Teller Machine) example ([Rumbaugh et al. 91] and [Wirfs-Brock et al. 90]). We demonstrate how COSEE can be used as a software development assistant, i.e. as a source of knowledge needed during software development.

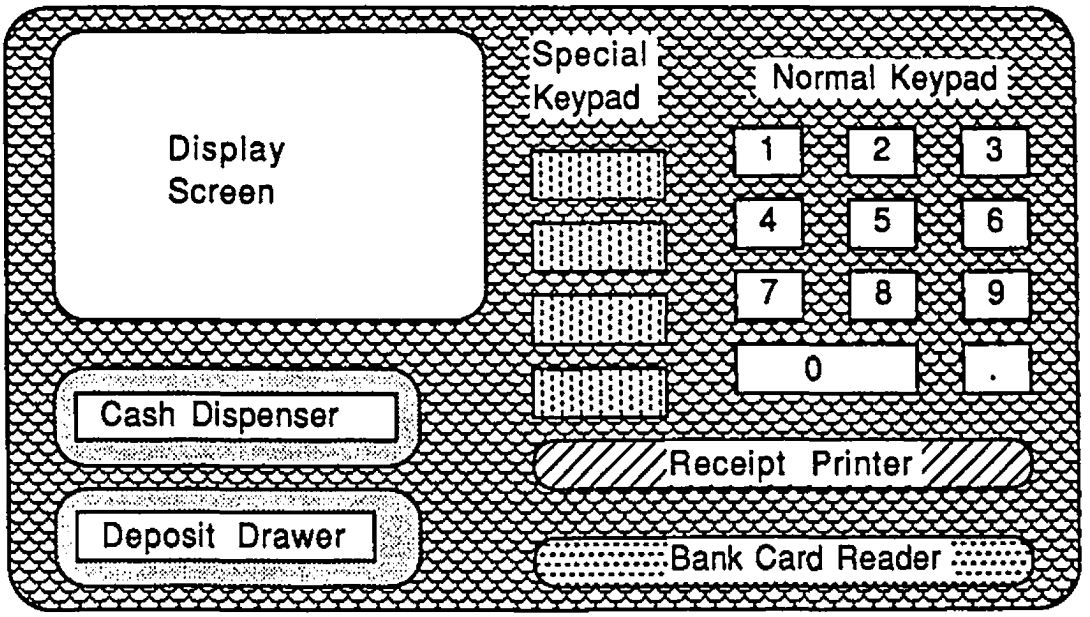

Fig 5.3 The ATM Structure

The knowledge in this example, about the ATM (Automated Teller Machine), is driven by the application domain, i.e. banking.

The ATM example illustrates our COSEE approach as follows:

- *The domain knowledge* is independent of any design and implementation, i.e. it is knowledge bankers can understand. We have only included knowledge relevant to the ATM.
- *The design knowledge* is independent of any implementation, i.e. it could be used for various implementations. However, it is built upon the knowledge required on the domain knowledge, and upon previous design components that are being reused.
- *The implementation knowledge* describes a particular implementation, i.e. it is language dependent (in this case Smalltalk-80).

For example, the concept of an account balance in the domain knowledge is captured as a concept understandable to bankers. In the design knowledge, it maps into the attribute "balance" of the

Account class. This in turn maps into an implementation in the case of Smalltalk-80 as an instance variable "balance" of the Account class. Hence from the domain, one can follow through to find out what happens to the idea of account balance once it is finally implemented; in this case it becomes an instance variable. Conversely, a programmer or a maintainer, if having difficulty understanding the idea behind the "balance" instance variable, could trace it back and understand how it fits into the domain knowledge, i.e. it corresponds to a bank account balance. Obviously in this simple example, there might not be much difficulty making this conclusion, but in much more complicated examples, where the domain is not familiar to the designers or implementors, making such inferences would be difficult or almost impossible. Thus system developers and maintainers can not only view the idea of a balance implemented in ST-80 environment but can look at it, at the same time, from the three COSE viewpoints as well. By having COSEE browsers (Domain knowledge, Design knowledge, and Implementation knowledge browsers) and ST-80 browser open at the same time, the user can select the desired concept (balance) in any browser and, by following the pointers, the corresponding concept will be selected automatically in the three other browsers (this what we call a "dynamic link") as shown in the following figure:

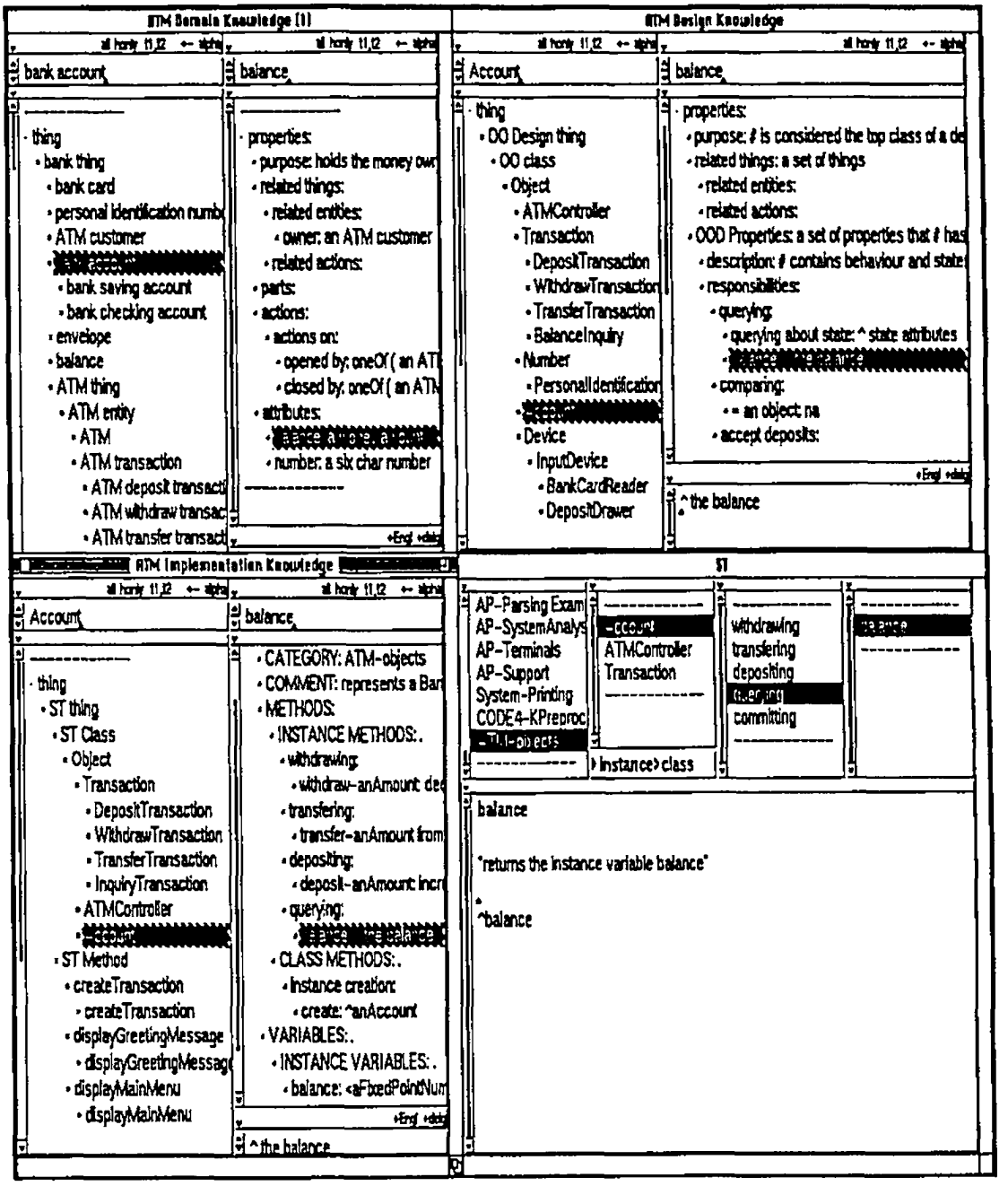

Fig 5.4 COSEE browsers: four browsers are dynamically linked. A selection 'bank account' in the domain knowledge drives selections in the other three browsers

5.2.1 ATM Domain Knowledge

In this section, we describe how we encode the ATM domain knowledge in COSEE. We discuss generic domain and application-specific concepts, the concept and property hierarchy, and our analysis methodology.

5.2.1.1 Generic Domain vs. Application-Specific Concepts

In analyzing the application domain of the ATM, we begin by trying to capture all necessary banking concepts in a knowledge base. A key point is that we define all the necessary concepts only in terms that a banker can understand, such as the bank card, the personal identification number, the bank account, the account balance, the ATM machine, the ATM transaction menu, the different types of transactions, and the different ATM parts. If there was previous banking knowledge (from an earlier project), hopefully some of it could be reused. The domain knowledge concepts can be categorized into generic (application-independent) and application-specific concepts, in this case the ATM described from the bankers point of view. We distinguish between these two kinds of domain concepts by tagging generic domain concepts (i.e. banking concepts) with two asterisks (**) as shown in the figure, i.e., these exist independently of ATM concepts.

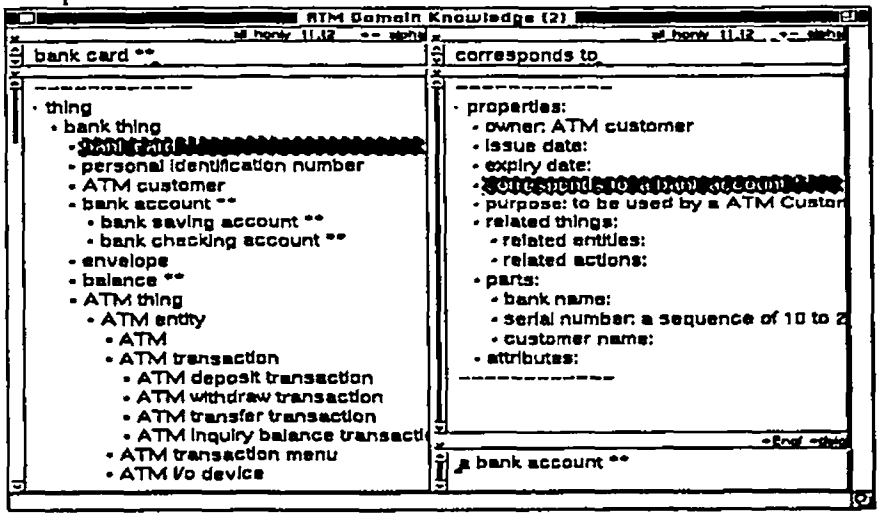

Fig 5.5 ATM Generic & Application-Specific Concepts

5.2.1.2 Domain Concept Hierarchy

We cannot describe all the domain concepts (or their properties) here; that is the purpose of the knowledge base. Basically, there are several important top-level concepts, which COSEE can easily display as in the following figure:

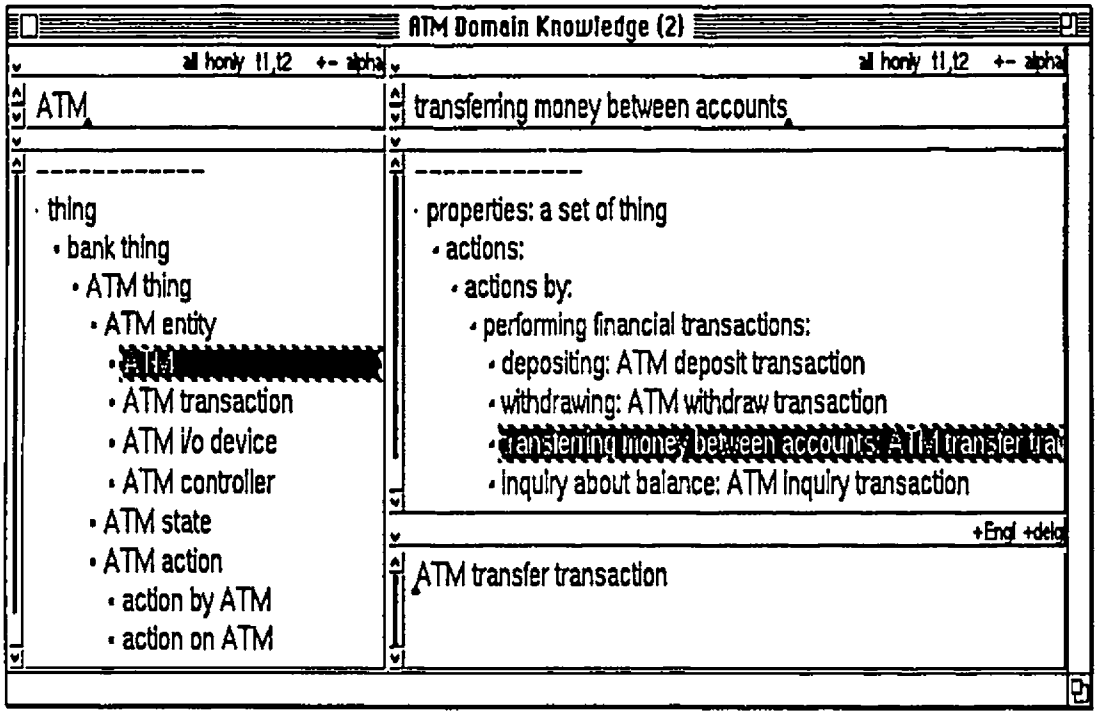

Fig 5.6 ATM Domain Knowledge - Top Level Concepts

We describe some of them briefly:

- **Bank thing:** any concept related to the banking domain or the application specific; i.e. that is not specifically described for the application only (such as bank card, bank account, and balance).
- **ATM thing:** any concept related to the application to be developed (such as ATM Controller, ATM state, and ATM action).
- **ATM action:** any action performed by (such as display, prompt, eject) or on (such as query) the ATM.

The following figure shows the complete hierarchy of banking concepts:

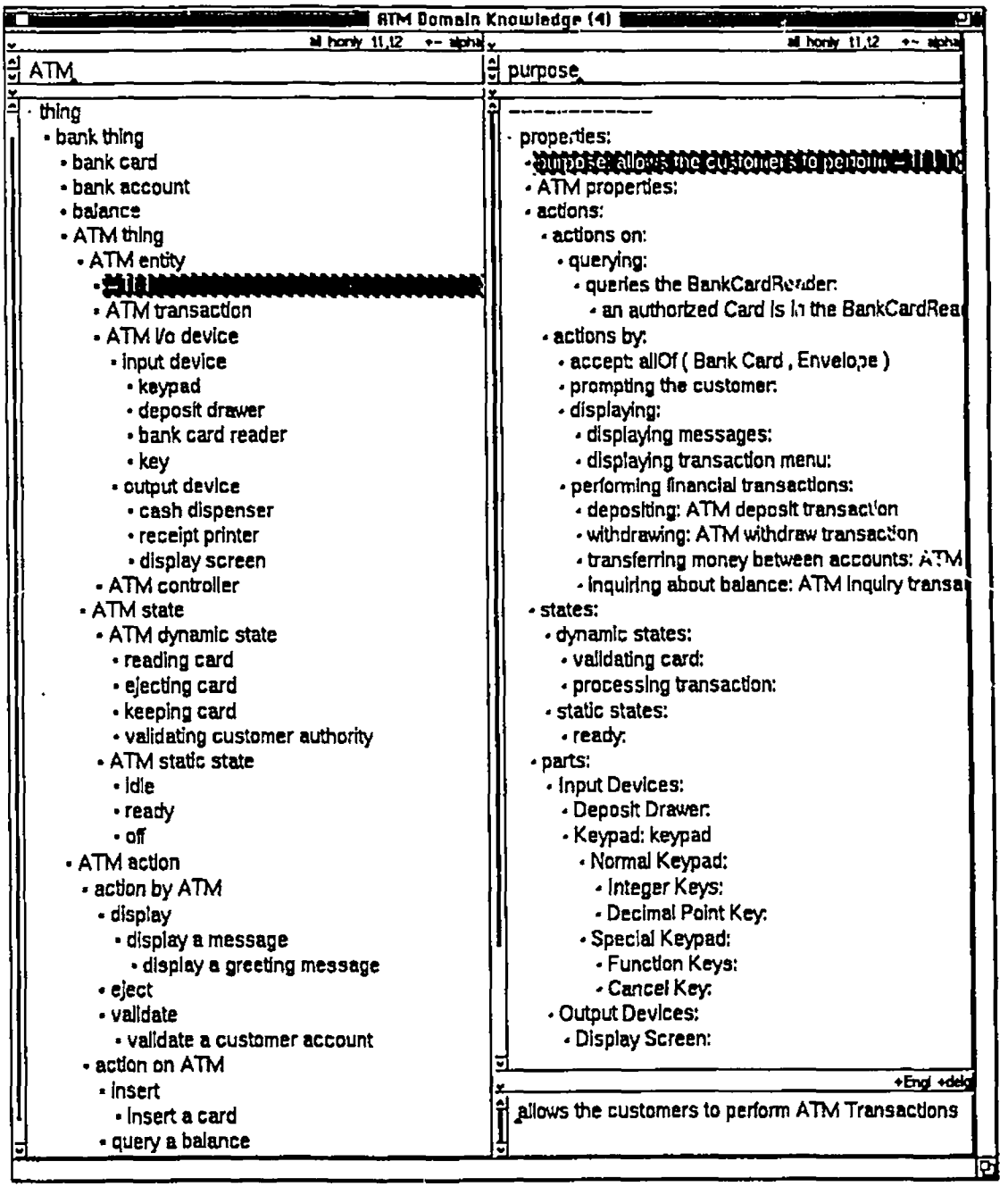

Fig 5.7 ATM Domain Knowledge - all concepts, with properties of ATM shown

5.2.1.3 Domain Property Hierarchy

Besides capturing banking concepts and problem-specific concepts in COSEE in a conceptual hierarchy, we can also describe some of their important and complex properties. Some of the properties of some concepts are themselves sufficiently complex that they ought to be treated as concepts themselves. For example, we listed, under the ATM, the actions and states as properties. However, we treat them also as concepts in themselves; a link in COSEE allows one to jump from a property such as "transferring money between accounts" to the associated concept that describes it in more detail. Two of the most important concepts that have to be understood in order to understand an application domain are the states and the actions, jointly called behaviour.

Like any concept, a banking concept is described by key properties, such as: purpose, related things, parts, behaviour. We describe these key properties in more detail:

- **Purpose:** What is the purpose of the concept, including its functionality in the system? For example, the purpose of the "Personal Identification Number" is "to identify the ATM user" and "provide a way for the ATM to check the authority of the user to access the system and perform financial transactions". Purposes are given by a short phrase using a simple verb.
- **Related things:** What are the different things that are closely related to a certain concept? For example, the two most closely related things for the bank card are the ATM customer and the bank card reader, i.e. to understand a bank card, you must understand these two concepts. The related things for a bank account are the personal identification number and the ATM customer
- **Parts:** What are the different parts of a banking concept (if any)? For example, the ATM has ATM Controller, Input Devices, and Output Devices. Input Devices consist of the Bank Card Reader, the

Deposit Drawer, and the Keypad. Output Devices consist of the Cash Dispenser, the Display Screen, and the Receipt Printer.

Using CODE's graphic capabilities, a part-of graph can be generated from the ATM domain concepts. By describing the different parts of the ATM, the bankers or the systems personnel (especially the designers) can benefit from a diagram showing the relationships between different components of the ATM:

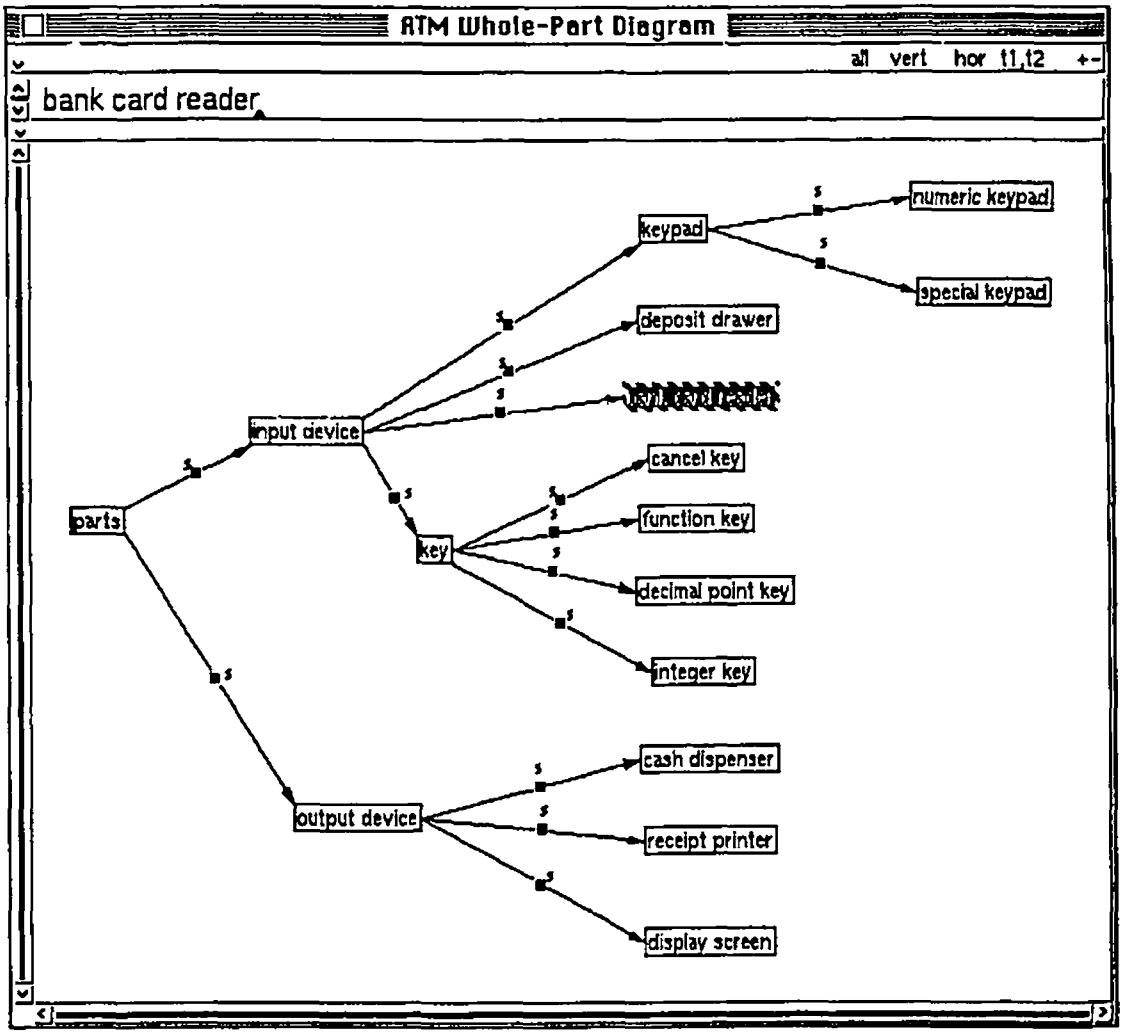

Fig 5.8 ATM Whole-Part Diagram

- **Behaviour:**

The behaviour is perhaps the most important property of a concept, at least for those that "have" a behaviour. It is usually the most difficult to describe and understand. From the domain point of view, the behaviour of an ATM can be described by states and actions in a simple finite state diagram, again, understandable to a banker. However, the designer will work from it and will add considerable details at the design phase, mostly corresponding to parts of the system that the bankers are unaware of.

Next, we describe the ATM states and then the actions performed on and by the ATM:

States:

We have come to the conclusion that the notion of state is quite complex and needs considerable analysis. For example, one major categorization we have detected is the difference between what we term dynamic and static states. By dynamic state, we mean a state in which the ATM remains while performing an activity or an action. By static state, we mean the state in which the ATM remains inactive waiting for an event to occur. A dynamic state is characterized by having an answer to the question "what is happening in the state?", whereas a static state is characterized by having the answer "nothing" to this question. For example, the ATM Controller can be in a dynamic state like completing transactions, validating ATM User authority, checking the User Account, etc. Or it can be in a static state like being idle, off, or ready for performing any transaction or accepting any user response. Using CODE4 graphics capabilities, we can draw the following ATM finite state machine diagram that we constructed following [Rumbaugh et al. 91] methodology:

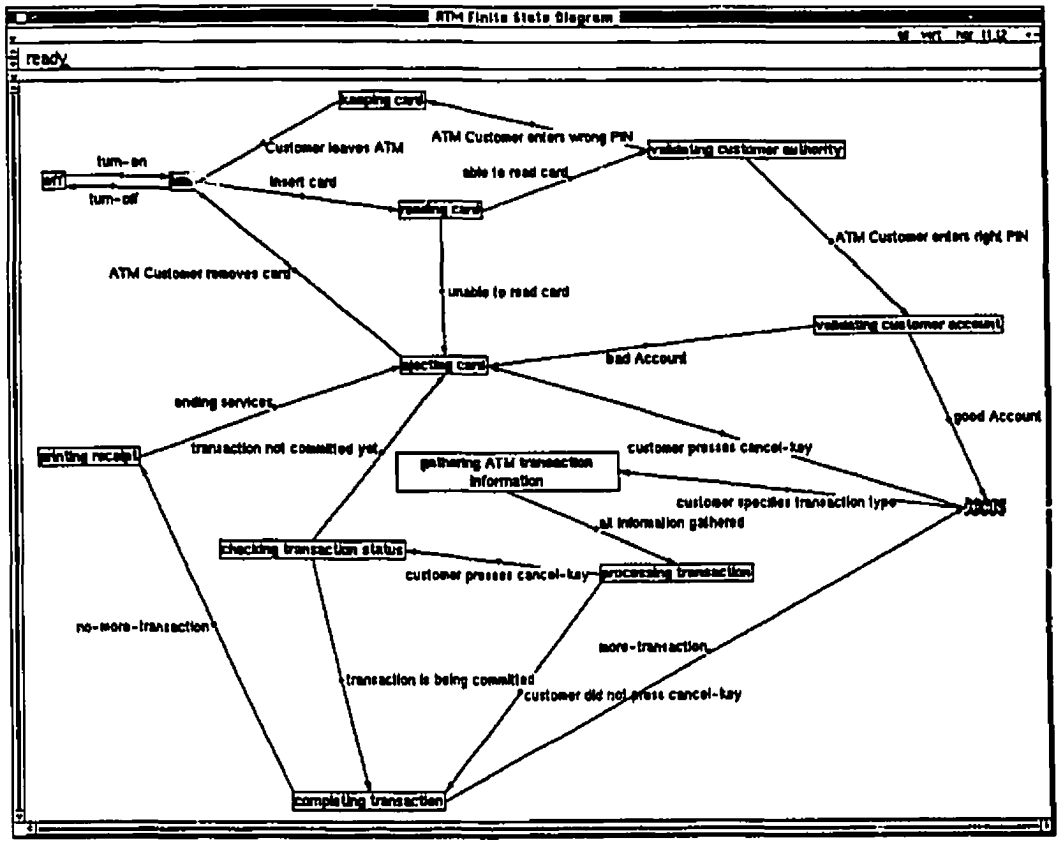

Fig 5.9 ATM Finite State Diagram: states are represented in rectangles while events appear on the links between them

In describing the ATM states, we differentiate between dynamic and static states. For each, we specify the state change event; what are the different events that change a specific state and what are its successor states. A finite state diagram shows the different ATM states and gives a good picture of the behaviour of the system that the banker can understand. Nerson [Nerson 92] explains the importance of the dynamic model: "The dynamic model consists of scenarios demonstrating significant object communication protocols. The purpose is twofold: it helps to validate the static model and to make sure objects are reachable from others; it maps the system

behaviour better as opposed to the static model that only reflects the structure". We could differentiate between dynamic and static states of the ATM by drawing them using different shapes (a feature that should be added to CODE4).

We now describe the events of the states by categorizing them into:

- **Incoming Events:** What are the input events to a state? For example, an incoming event for the ATM state "reading card" is "insert card".
- **Outgoing Events:** What are the output events from a state? For example, the outgoing events from the ATM state "reading card" are "able to read card" and "unable to read card".

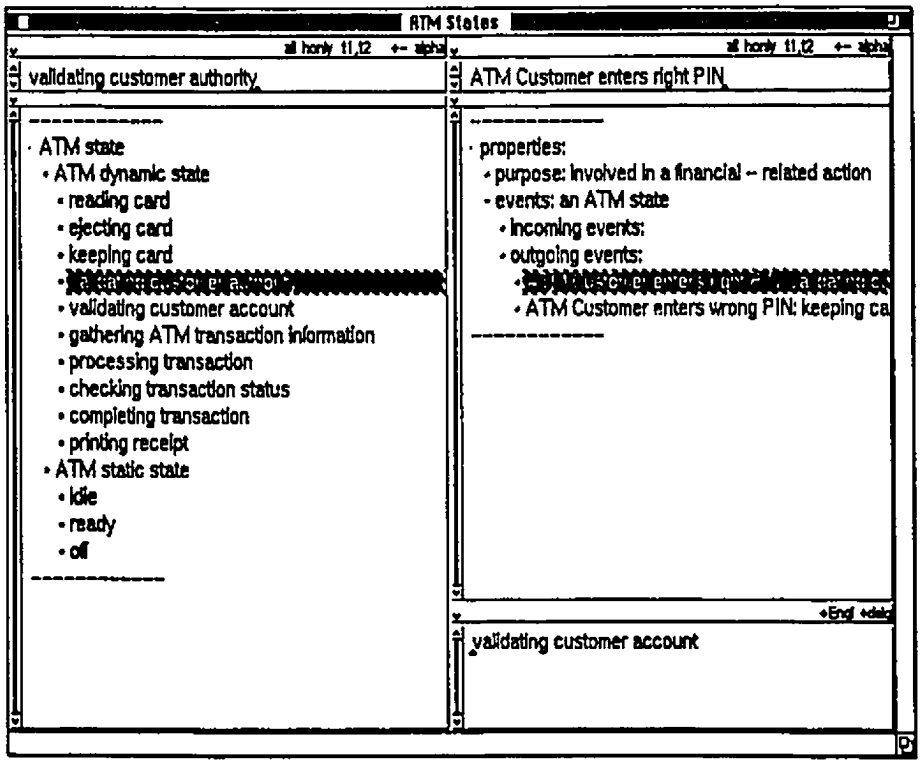

Fig 5.10 ATM states

Actions:

By actions, we mean the different actions performed either on or by a thing. For example, the Actions performed On the ATM Controller include 'querying'; the ATM user queries the Controller about his/her account balance. The Actions performed By the ATM Controller include 'displaying'; the ATM Controller displays messages (like greeting, inserting-card, and removing-card messages) to the ATM user. In describing the actions performed on and by the ATM, we specify the following properties:

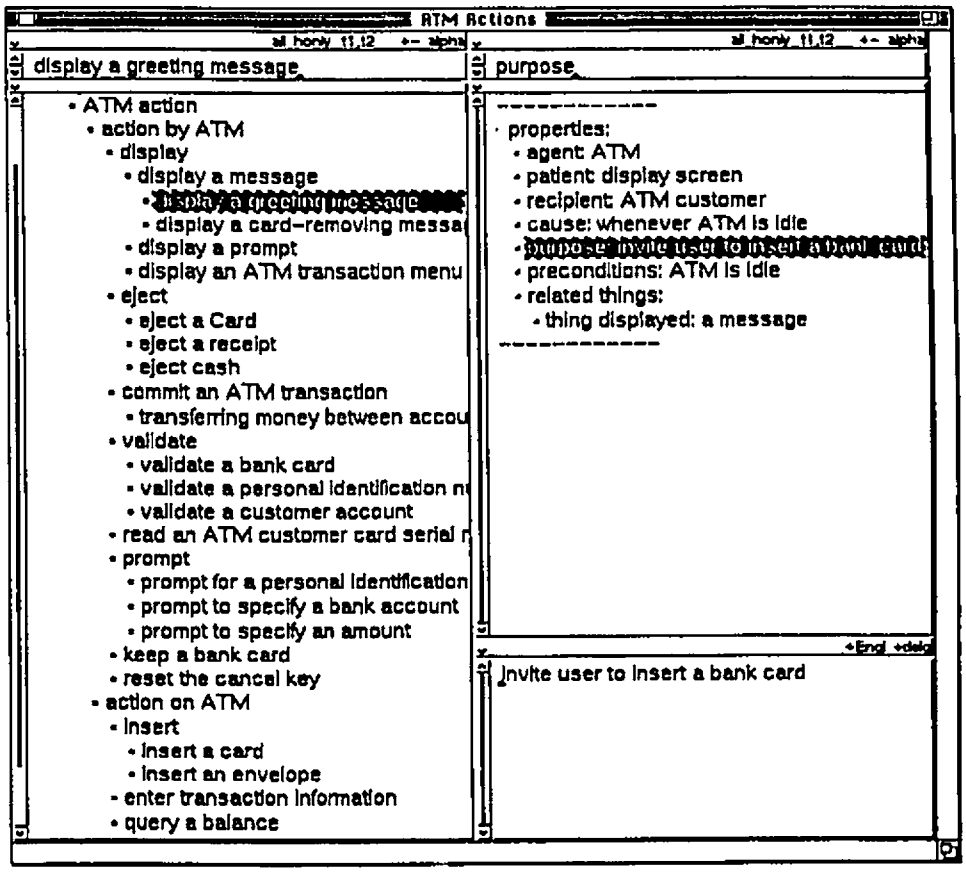

Fig 5.11 ATM Actions

- **Purpose:** What is the purpose of performing a certain action and what is its effect on the ATM in general? For example, the purpose of 'displaying a greeting message' is 'invite user to insert a bank card'.
- **Agent:** Who is the agent of the action? (Who or what does it) For example, the agent of 'displaying messages' is the ATM Controller.
- **Patient:** Who or what is the patient (i.e. is affected by) of the action? The Display Screen is the patient of 'displaying messages'.
- **Recipient:** Who is the recipient of the action? The ATM User is the recipient of 'displaying messages'.
- **Cause:** What is the cause of a certain action? The ATM Controller asks the Bank Card Reader to keep the Bank Card because the ATM User fails to enter the right personal identification number.
- **Pre-condition:** What is (are) the pre-condition(s) in performing a certain action? Before the ATM User inserts a Bank Card in the Bank Card Reader, the ATM Controller must displays a greeting message inviting the ATM User to insert a card and the Bank Card Reader must have a space for the new card.
- **Constraints:** What are the constraints on the action that is to be performed? For example, the ATM User cannot withdraw or deposit more than a certain amount each day.
- **Related entities:** What are the entities related to performing a certain action? A related entity to entering the 'personal identification number' is the User Account.
- **Related actions:** What are the actions related to performing a certain action? A related action to the action of 'completing a certain transaction' is to ask the ATM User if other transactions are needed to be performed.

Analysis Methodology

Improving analysis was not our objective; we relied heavily on [Rumbaugh et al. 91] and [Wirfs-Brock et al. 90]. We started by describing banking concepts in natural language, using COSEE as a blackboard for rapidly capturing ideas that would arise during discussions with the domain expert in a knowledge base regardless of their formality and their sequence. We augmented what we found in [Rumbaugh et al. 91] and [Wirfs-Brock et al. 90] with our own knowledge about banks and ATMs. Another way of capturing these initial concepts in the knowledge base would be from existing documents (CODE4 has a facility, under development, for reading documents and extracting concepts from them). Specific system requirements are identified in the knowledge base as concepts and properties; the analyst can attach flags to them in CODE4. For example, interview questions prepared by the analyst might be stored in the knowledge base until the domain expert answers. When the analyst wishes, the knowledge base can be presented to the domain expert to check and validate its contents.

The analyst could benefit from the predefined ontology available in CODE4 which presents a taxonomy of generic high level concepts with their generic properties. Our methodology benefited from the ontology, just as the Smalltalk-80 programmer benefits from the built-in classes. This ontology plays an important role in getting the domain experts and the developer team to agree on common terminology. Also, systems analysts can more easily reuse another knowledge base (or part of it) that describes the same (or similar) domain knowledge if it has the same top level ontology. This knowledge base can be loaded in memory and the systems analyst can copy and paste the required knowledge between knowledge bases. The top-level ontology describes very generic concepts like state, action, activity, event, process, etc. in a unified manner so that everyone can be in agreement on the meaning of these terms. For example, an ATM state can inherit generic properties from the description of the state and the systems analyst adds specific

properties (like those previously discussed) to it.

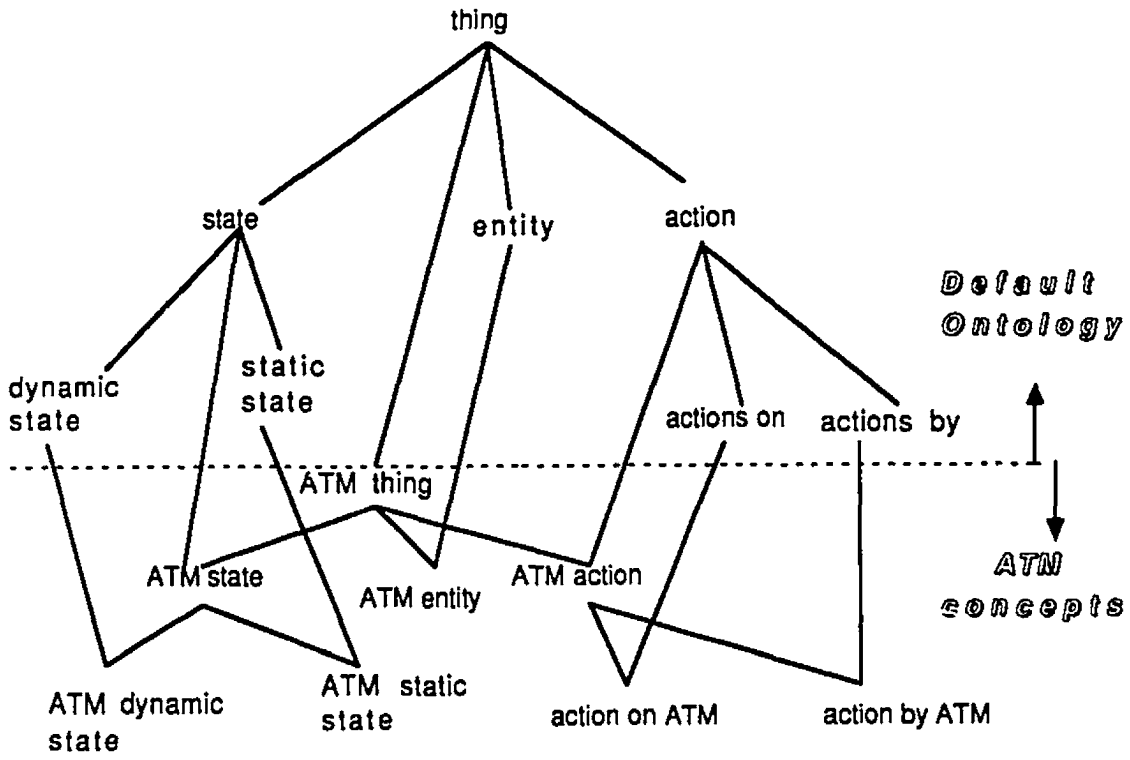

Fig 5.12 Relation of ATM Concepts to Default Ontology in CODE4

The next step to narrow the gap between the analysis and the design phases is done by the systems analyst by identifying an initial list of domain objects. This is done by flagging candidate concepts in the knowledge base and by highlighting the system requirements in terms of concepts and properties. For example, the analyst could flag such concepts as: The ATM, the account, the personal identification number, the input and output devices and flag such requirements as deposit, withdrawal, transfer, and balance inquiry. To differentiate between system concepts and system requirements, the analyst could use different notations.

5.2.2 ATM Design Knowledge

Starting from a set of selected domain concepts (potential object-oriented classes), the systems designer can begin to determine the candidate classes in the knowledge base in an object-oriented hierarchy under the concept "OO design thing". Since we are using an object-oriented approach in our design, the hierarchy contains two main subhierarchies: One corresponds to classes (under the concept "OO Class") and the other corresponds to class behaviour (under the concept "OO Behaviour"), which will be implemented by methods.

OO Class Subhierarchy:

Under the "OO Class" subhierarchy, the designer specifies the different ATM classes with their purposes, their descriptions, their states, their behaviours, and their design rationale. For example, the Transaction class would be described with the following properties:

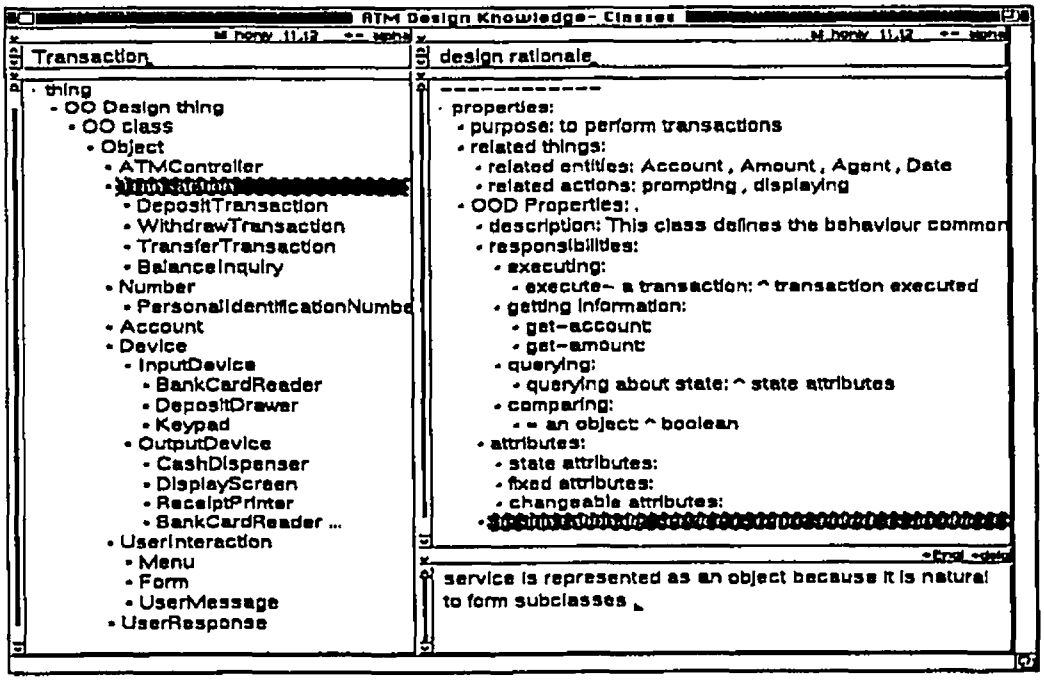

Fig 5.13 ATM Design Knowledge - OO Classes

- **Purpose:** To allow the ATM User to perform a certain financial transaction.
- **Description:** Class Transaction defines the structure, related things, behaviour, etc. common to all requests from a bank customer to perform a financial transaction.
- **Responsibilities:** The responsibilities (behaviour) of the Transaction class can be grouped into: Executing and Private. Executing includes "execute a financial transaction". Private includes those private responsibilities for the class like prompting for an account or an amount, and committing the transaction to the database.
- **Attributes:** The attributes can be divided into state attributes (which define the state of the object), fixed attributes (attributes that cannot be changed), and changeable attributes (attributes that can be changed but are not 'state'). For example, a transaction can be in one of the following states: completed, uncompleted, canceled, waiting, or suspended. It also has a fixed attribute such as the account and a changeable attribute such as the agent of the transaction.
- **Design Rationale:** The designer explains why a service like the transaction has been treated as a class and not as a behaviour of another class: To form subclasses (the different kinds of transaction) from the class Transaction. And why this class is designed to be a subclass of the Object class: The actual superclass will be left until the implementation phase; since it is dependent on the built-in classes of the language (here, the Smalltalk-80 language).

OO Behaviour Subhierarchy:

Under the "OO Behaviour" subhierarchy, we encode the different behaviours of the ATM system as concepts and attach properties to

them in order to help the programmer in its coding and to facilitate the building of our implementation knowledge. The key properties attached to each behaviour are:

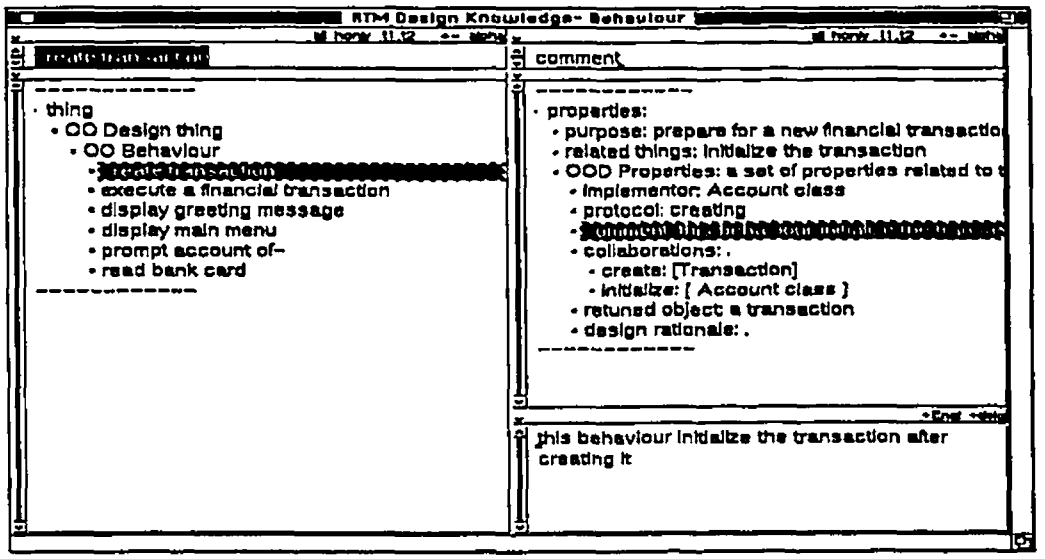

Fig 5.14 ATM Design Knowledge - OO Behaviour

- **Implementor:** Who is (are) the implementor(s) of the behaviour? For example, the behaviour: "execute a financial transaction" is first implemented by the Transaction class and modified by its subclasses (which are the different kinds of transactions).
- **Purpose:** What is the purpose of introducing the behaviour for each implementor? For example, "execute a financial transaction" is inherited from the class Transaction to the class Deposit Transaction (or Withdraw Transaction) to allow the ATM User to perform a deposit (or a withdraw) transaction.
- **Comment:** Any comment related to the behaviour. For example, the designer can make a point that this behaviour might be extended to perform more tasks or the behaviour can be achieved by sending different messages to other objects than those mentioned.
- **Collaborations:** What are the different messages sent from the

implementor to other objects in order to achieve a certain behaviour? For example, in order to display the balance of a User Account, the class Balance Inquiry Transaction must collaborate with the classes: ATM User, Account, and Display Screen by sending them appropriate messages.

- **Returned object:** What is the value (or the object) returned by the behaviour after a service is completed? For example, when the Account class is asked if it is valid, it returns a boolean value indicating whether it is good or bad account.

As we have described the different ATM states in the domain knowledge, we can add a subhierarchy describing the states that are important to the designer, such as the ATM state "gathering information", the communications between the ATM controller and the other parts of the machine, committing the information to the database, database files maintenance, and so on. Our diagram is similar to the state diagram of [Rumbaugh et al. 91]. However, the latter mixes states and events but our diagram is simpler, more readable, and more expressive; we represent states as nodes and events as links. An example of describing a dynamic state is the "Validating Customer Authority": The events changing this ATM state are that the ATM User either enters the right or the wrong personal identification number. In the first case, the next ATM state is "Validating Customer Account" and in the second case, the next ATM state is "Keeping Bank Card".

OO Design Methodology:

The design we present follows more or less th design given by [Wirfs-Brock et al. 90], however it could be easily changed to follow the design given by [Rumbaugh et al. 91]. We extend this design by allowing the designer the identification of the requirements specification and the system functionality by tagging the behaviours of the correspondent classes.

5.2.3 ATM Implementation Knowledge

In this section, we discuss how we represent the different types of knowledge about the coding itself, i.e. that are needed by the programmers and the maintainers. When the programmer originally creates Smalltalk-80 classes and methods, his/her responsibility is to encode knowledge about them in COSEE knowledge base. This will help other programmers and maintainers understand the code. Since it is not one of our goals to produce an implemented ATM system, we have devoted the major part of our emphasis on the domain knowledge and the design knowledge.

One of our research goals is to try to capture in COSEE all the knowledge about Smalltalk-80 classes and methods, except the code itself. The purpose is to relieve the programmers and the maintainers from the difficulty they experience when they try to understand the system from the code itself. They do not have to open a Smalltalk-80 browser except for inspecting the code. They open the implementation knowledge browser with a dynamic Smalltalk-80 browser so that whenever they select a class or a method from the implementation knowledge browser, the corresponding selections are done automatically in the Smalltalk-80 browser, and vice versa.

The classes in the design knowledge may not have to one-to-one in correspondence with the classes in the implementation knowledge. A class in the design knowledge might have many corresponding classes in the implementation knowledge and vice versa. For example, the class ReceiptPrinter in the design knowledge might correspond to the classes ReceiptLinePrinter and ReceiptLaserPrinter in the implementation knowledge. And the classes ATM-IO-Device, ATM-InputDevice, ATM-OutputDevice in the design knowledge might correspond to the class ATMInputOutput in the implementation knowledge.

Implementation Classes:

Under the "ST Class" subhierarchy, the programmer specifies the different ATM classes with their purposes, their category, their protocols, comments about them, their methods (instance and class methods), and their variables (instance and class variables). Thus, for every class, we attach the following key properties (some of them are represented in the figure in uppercase to point out that they are extracted directly from the Smalltalk-80 environment) :

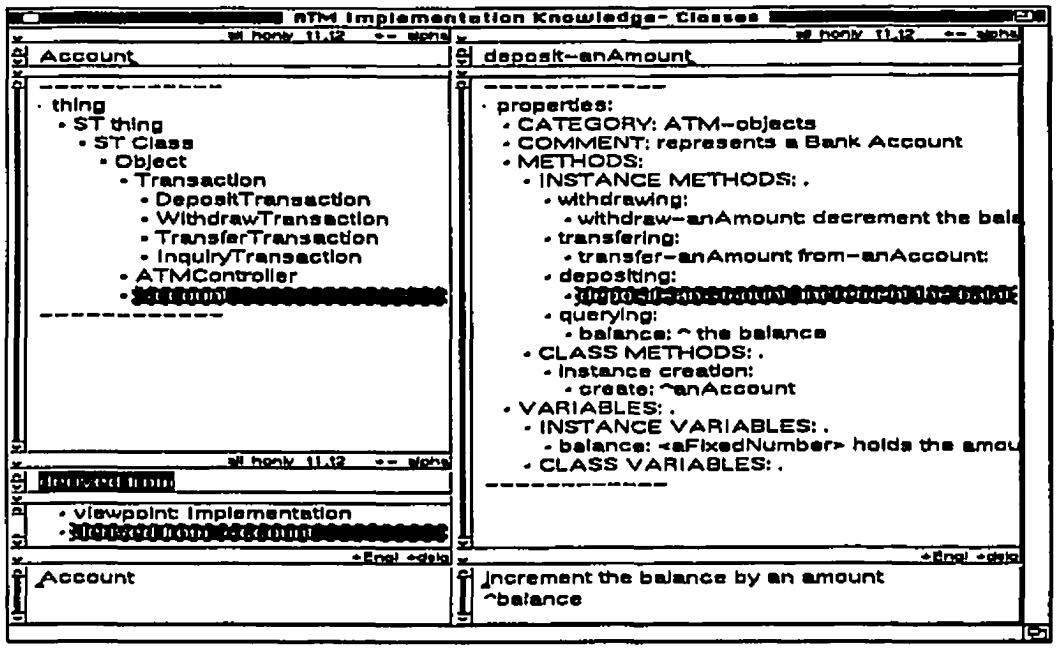

Fig 5.15 ATM Implementation Knowledge - ST-80 Classes

- **Category:** Under which category will the class be classified? For example, a new category called "ATM System" might be created in the ST-80.
- **Protocols:** What are the protocol names for the class? For the class ATM Controller, we created a protocol called "displaying messages" under which we group all kinds of displaying messages (displaying a greeting message, displaying a removing-card message, displaying a keeping-card message, etc.).

- **Instance Methods:** What are the instance methods that every instance of the class must have? For example, every deposit transaction will be an instance of the class Deposit Transaction.
- **Class Methods:** What are the methods that apply to the class as general and not to its instances? The creation of transaction instances is an example of such methods. For each method, one can easily see the selector, argument types, and class of returned value.
- **Instance Variables:** What are the instance variables of the class? These variables include the states and any other variables needed for the implementation, which were distinguished as different kinds of attributes in the design. The Balance is an instance variable of the class Account.
- **Class Variables:** What are the class variables that all instances of the class will share? For example, a certain checking constant code might be required before the ATM User enters the personal identification number. This code is implemented as a class variable for the class PersonalIdentificationNumber.

Implementation Behaviour:

The behaviour is described in terms of ST-80 methods, the states in terms of instance variables. The purpose of a behaviour given in the design knowledge remains the same for the implementation knowledge. Every description of a class behaviour in the design knowledge is linked to the implementing method. Method descriptions include details such as temporary or instance variables used within the method, or the purpose of every collaboration (so that the programmer or the tester can check the correctness of every implemented collaboration and in turn the implemented method). The goal is to eliminate, as much possible, the need for recording actual code. The user has many choices on how to select or display group of related methods.

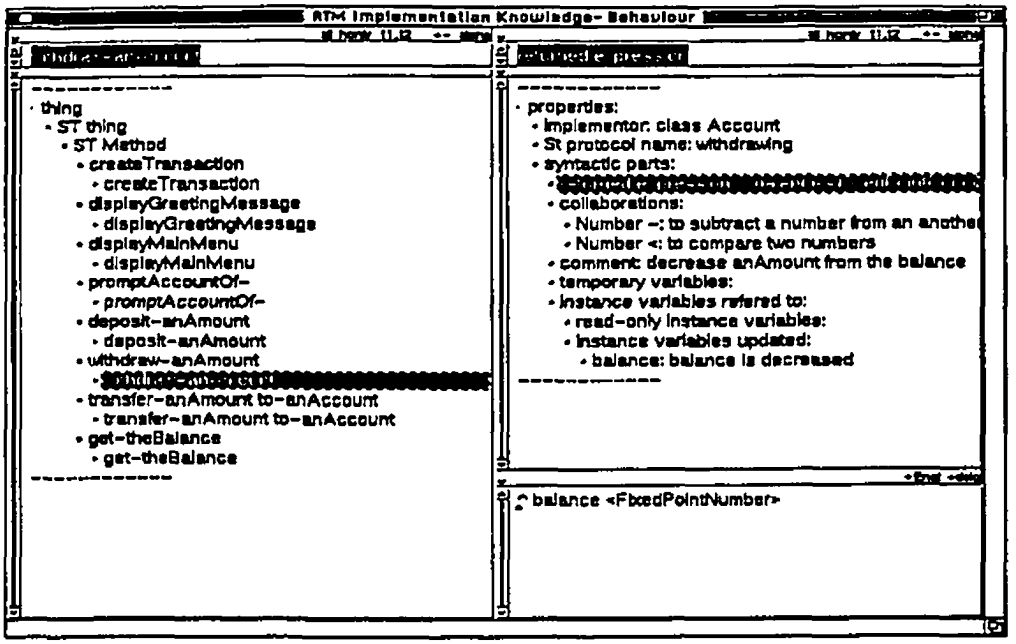

Fig 5.16 ATM Implementation - Behaviour

(N.B.: We use '-' in the selector instead of ':' to avoid confusion with the ':' of the property attached automatically by CODE4)

For example, the method 'withdraw-anAmount' would have the following key properties:

- **Implementor:** Who is the class implementor? The class Account.
- **St-protocol name:** What is the ST-protocol of the method? 'withdrawing'.
- **Syntactic Parts:** What are the different syntactic parts of the method?
  - **Returned Expression:** What is the returned expression? The balance, which belongs to the class <FixedPointNumber>.
  - **Collaborations:** What are the different messages sent to other classes? We list all messages sent form within the method. The convention for the property name is <receiver class> (if known) followed by the selector. The convention

for the value is to copy the purpose from the descriptor for the method.

- The message < is sent to the class Number to compare balance to anAmount.
- The message - is sent to the class Number to subtract anAmount from balance.

- **Comment:** What are any other comments that the programmer want to make (normally begins with the purpose)? Decrease the balance by the amount 'anAmount'.
- **Temporary Variables:** What are the different temporary variables used in the method and their purpose? none.
- **Instance Variables Referred to:** What are the different instance variables referred to (used) in the method and their purposes?
  - **Read-Only Instance Variables:** What are the instance variables that have not changed their values? none.
  - **Updated Instance Variables:** What are the instance variables that are affected by the method?
    - **balance:** the balance is decreased.

Implementation Methodology:

Using a COSEE design knowledge base, the programmer "plugs" the ATM classes among the ST-80 classes as usual. In practice, all existing ST-80 classes and methods would be described in a knowledge base the programmer begins with; making it easier to find and understand appropriate classes for reuse. However in our example, only relevant existing ST-80 classes are shown in the implementation knowledge. For example, the implementation knowledge shows the ST-80 class Number because PersonalIdentificationNumber is specified as a subclass.

Starting from a knowledge base that contains the three COSE viewpoints (the domain, the design, and the existing implemented classes), the programmer (the Smalltalk-80 coder) has a relatively small amount of work to do (than without COSEE). The programmer's task is to code the classes and methods in ST-80, at the same time capturing knowledge about each new one in COSEE. Browsing and thinking with COSEE should precede the actual coding. Having the implementation knowledge browser and a ST-80 browser open, the programmer can do the job more quickly by directly adding another layer of detail on top of that provided by the existing implementation knowledge.

If the programmer needs to further understand the purpose, the design rationale, or any comment related to a specific class or a specific method, he/she can open the design knowledge browser. For example, the programmer might want to know why the class PersonalIdentificationNumber is designed as a subclass of the class Number and not of another class that he/she might prefer. To understand the description of a certain concept, the programmer can open the domain knowledge browser (background knowledge) instantly . If he/she wishes to have four browsers open at the same time: The ST-80 browser, the implementation knowledge browser, the design knowledge browser, and the domain knowledge browser, the programmer can trace a certain class from the ST-80 implementation all the way back to the domain knowledge. This can be done by making a selection in any browser; and the corresponding concepts in all the other three browsers would be automatically selected by COSEE.

After the programmer implements the different classes and methods in the ST-80 environment, a simple mechanism can check that all classes and methods in the implementation knowledge have been mapped into the ST-80 environment. A list of all classes and methods that have not yet been mapped can be generated from the system.

5.3 Features of Knowledge Management Systems Useful for Software Developers

Ideally, a system developer expects features in a development environment that provide as much assistance and guidance as possible. Today's development environments tend to assist the developers on details of the programming process or, at the design end, assist by making certain types of analysis techniques (such as entity-relationship diagrams or dataflow diagrams) easier to do by extensive graphical aids. However, since they lack any knowledge engineering features, both in knowledge representation and user interface, they do not assist the developer very much in understanding concepts. This is of course the main purpose in the research we are undertaking; to emphasize the importance of an environment that can both represent and assist in understanding the kind of conceptual and descriptive knowledge that are needed to understand the various stages of the development process. This is done at two levels: the knowledge representation level and the user interface level.

In this section, we will explain what are the knowledge representation features and the user interface features that a development system should possess for our view of knowledge management. We will also explain what are the features that now exist and the ones that need to be added in CODE4 and COSEE.

5.3.1 Knowledge Representation Features

In this section, we will present the main knowledge representation features implemented in CODE4* . In our experience, these features are sufficient to be used in software engineering, hence we haven't added or proposed other knowledge representation features in CODE4 or in COSEE.

---

* Section 5.1.1 introduces the basic concepts.

For our purposes, we will highlight the most important knowledge representation features that are needed for software engineering and how users can make use of these features in CODE4 and in COSEE:

- The ability to express knowledge in different forms without being limited to a specific form of a representation:

  In CODE4, the user can make statements about concepts, i.e. types and instances. The user can enter knowledge in an unconstrained natural language, if desired, and still do some useful knowledge management with it, or can use more formal expressions. CODE4 can be as simple as an outline processor or as complex as a first-order logic system.

- The ability to organize knowledge in a way similar to the way people manipulate knowledge:

  In CODE4, knowledge can be organized in hierarchies, which is a very natural way for people to think.

- The ability to have some form of property inheritance for the concepts in the knowledge base; i.e. the user does not have to describe the properties of a concept again if they are described in its parent concept:

  In CODE4, concepts are arranged in hierarchical conceptual definitions or descriptions based on the notion of inheriting properties to be the most useful and frequent knowledge organizing activity for the applications in use.

- The ability to perform some inferencing on the knowledge encoded in the knowledge base; i.e. to infer new knowledge from existing knowledge and to perform some checkings and validations:

  Two kinds of inferencing exist in CODE4: fast inferencing

(that can be done without compromising the expressiveness of the user) and slow inferencing (that could be done only on demand in the background and that could handle complex cases).

- The ability to attach incremental statements about statements; i.e. make statements about other statements:

  CODE4 has *facets*, i.e. properties of statements. Users can add their own or use the built-in facets (such as modality, status, statement comment, etc...). For example, modality is a facet that specifies whether the property is necessary, typical, optional, inappropriate, or false. For example, in the ATM example, we could say that a bank customer "typically" has a bank card.

- The ability to express properties, for a certain concept, that do not inherit to instances:

  CODE4 has the *metaconcepts* feature that is used to express knowledge about the concepts themselves; i.e. property values do not inherit to sub-concepts. For example, in the ATM example we could attach the metaconcept *class name* to every concept in the domain knowledge to act as a pointer to the corresponding class in the Smalltalk-80 environment (the account concept points to the class name Account in the Smalltalk-80).

- The ability to have some support for natural language:

  The representation in CODE4 allows the user to use ambiguous terms, synonyms, concepts without names, and to rename things. CODE4 has a facility for the treatment of linguistic knowledge, in particular, terminology; i.e. the association between concepts and phrases that denote them. CODE4 uses separate concepts to represent terms, i.e. that encode properties of the term, rather than the

concept itself. For example, in the ATM example, we could associate the following term properties with the 'bank' concept:

term: bank	
term of:	bank (i.e. a back pointer to the bank concept)
synonyms:	financial institution
meanings:	(pointers to other concepts named by this term, e.g. river bank)
part of speech:	noun
plural:	banks
french equivalent:	banque

Another feature in CODE4 is the ability to make terminological inferencing. For example, the statement "increase the amount" could be recognized as equivalent to "increase the balance", since these are synonyms.

- The ability to look at the knowledge from different perspectives:

  In CODE4, the user can label with a perspective name any concept or property in the knowledge base. The system can then show only those concepts or properties that belong to that perspective. (This acts as an additional grouping mechanism, orthogonal to the hierarchical grouping we have used to structure our three viewpoints).

5.3.2 User Interface Features

A knowledge representation system is not very useful without a good user interface. The more powerful a system becomes, the more important the user interface capabilities become. The importance of the user interface stems from the role it plays in the flow of knowledge between the user and the system.

In designing a user interface, the designer should take into consideration human abilities and who the intended users are; i.e. the user's limitations and strengths. A user interface's main function is to provide assistance to the user in accessing and structuring the information in the system. If the information has a hierarchical structure, then a hierarchical display is a natural way for people to manipulate knowledge either textually or graphically. Special symbols and notations can be used to remind the user of stored knowledge and improve the communication effectiveness.

We will classify user-interface features generically into those intended to assist in knowledge acquisition and those intended to assist in knowledge retrieval. By *knowledge acquisition*, we mean obtaining knowledge from the user (editing, reasoning, structuring, browsing, refining, brainstorming, etc...). By *knowledge retrieval*, we mean helping the user to extract and review existing knowledge (e.g. browsing, masking, diagramming, graphing, searching, cross referencing, etc...). Most of the features that we will discuss are essentially knowledge retrieval features; they assist the user in reviewing the knowledge in one form or another or restricting what he/she sees. However, knowledge retrieval is also an important activity during knowledge acquisition; frequently during knowledge acquisition, the user's need is not necessarily to add new knowledge but to look at the existing knowledge to decide what to do next (It is for this reason that we have listed the feature 'browsing' as being a feature for both knowledge acquisition and knowledge retrieval). Some features are specifically intended for knowledge acquisition; e.g. adding/deleting a concept, adding/changing or deleting a value to a property or restructuring the knowledge base in some way. However, these features are relatively small set of the total number of the user interface features needed in a knowledge management system, and are normally designed to work with the retrieval features (for example, in CODE4 a subwindow browser uses ctrl-a for easily adding a new subconcept of a concept in the hierarchy).

Two important concepts in CODE4 need to be defined before discussing the user interface features:

1) *The knowledge map:* A specification of a network of relations. Typ< , of knowledge maps include is-a hierarchies, property hierarchies, etc. Knowledge maps are treated as directed graphs and are displayed in subwindows of browsers.

2) *The knowledge mask:* A filter that determines whether a concept will be included in a knowledge map (and thus displayed to the user). It contains a logical expression relating a set of boolean conditions that are applied to each concept. Masks control the visibility of concepts and properties; they are used for hiding specific sets of concepts, as well as more detailed patterns of knowledge. Each knowledge map is defined by a knowledge mask.

We will highlight the most important user interface features that are needed for software engineering and describe how users can make use of these features in CODE4 and in COSEE:

Existing User Interface Features in CODE4

- Diagramming or graphing capabilities provide the user with another conception of the knowledge: A diagram presents the relations between different components using different layers of details. For example, concept hierarchical diagram, part-whole diagrams, state-transition diagrams, data flow diagrams, etc...:

  In CODE4, there are many types of diagrams such as: The concept (is-a) hierarchy, the property hierarchy, the relation hierarchy (e.g. state transition, part-whole diagram,...). For example, as we did in section 5.2.1, we can draw the domain concepts, the finite state diagram, the different parts of the ATM.

- The ability to assist the user to create and manipulate knowledge bases quickly; (brainstorming or sketching):

By making use of hot keys in CODE4, the user can create and manage knowledge quickly. For example, ctrl-a allows the user to create a concept (property) child, ctrl-d for concept (or property) deletion, ctrl-b then ctrl-p for reparenting. Also, the user can change any of these hot keys for his/her own purposes.

- The ability to help the user find knowledge quickly; i.e. the "Navigation" or "Browsing" feature: Browsing can be done either textually or graphically:

  In CODE4, textual browsing allows the user to display relationships in hierarchies by indentation while graphical browsing displays the relations as nodes-and-links allowing the user to arrange and re-arrange them in different ways. Functionality between text and graphic modes is made as consistent as possible. Both types of browser permit rapid hypertext-like navigation in and viewing of large hierarchical structures with multiple inheritance. The user can switch between different browsers at any time; windows are dynamically updated. Browsers are used to view and manipulate portions of a knowledge base. Each browser is composed of one or more subwindows. Each browser subwindow has an interaction paradigm; it displays knowledge either as a graph, an outline processor, a user language (simple text input by the user), or a matrix (like a spreadsheet). Where possible, however, operations are done in the same way, regardless of what interaction paradigm a subwindow is using. User can select/deselect single or multiple nodes or links or subhierarchies. Different commands can be performed from within the browser subwindows with menus, action buttons, or hot keys. Searching and replacing has an impact on the overall system productivity and effectiveness; it reduces the time spent by the user in locating and updating knowledge.

- The ability to detect and warn about problems and possible contradictions:

  In CODE4, the system warns the user if he/she attempts to delete a property that has a value (i.e. a statement). CODE4 could interactively check with the user if there are different terms for the same concept or one term used for different concepts. Contradictions could be detected, off-line, by exporting the knowledge base to a first-order logic subsystem or other formal language system. Associated with each knowledge map is an optional feedback panel that displays a list of attempted commands, command results, suggested actions, and suggested commands that help the user solve the problem. CODE4 displays in the feedback panel suggestions in response to any user action that does not seem consistent or reasonable.

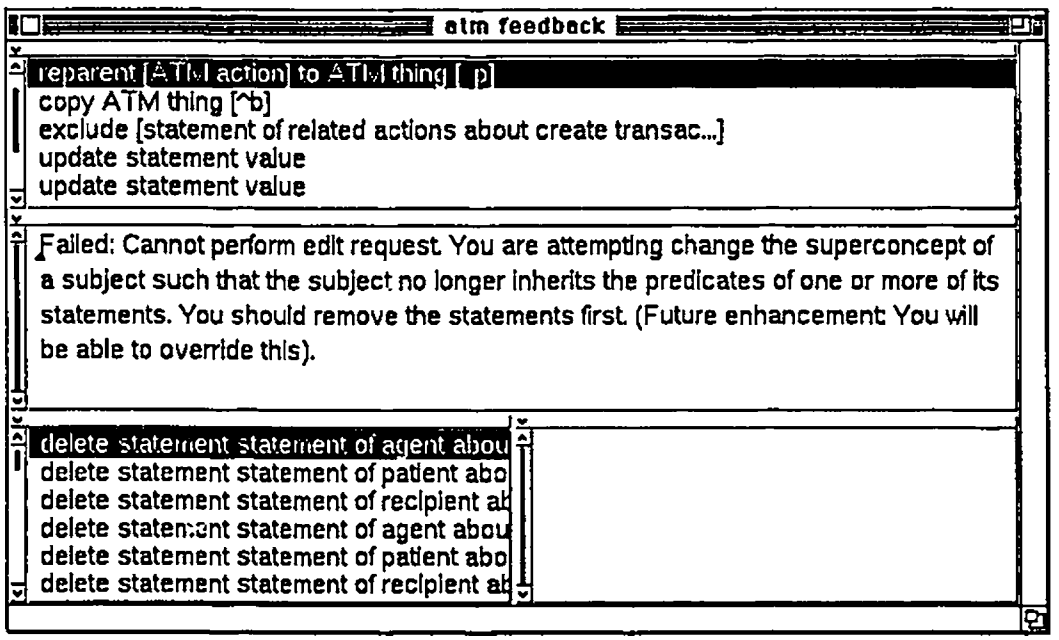

Fig 5.17 ATM Feedback Panel: an attempt to perform an operation (on the concepts) with a warning to the user that it might cause a conceptual problem. Possible "cures" are listed in the lower left pane.

- The ability to allow the user to filter or mask knowledge according to some criteria to reduce the amount of information visible:

  In CODE4, associated with each knowledge map and with its browser subwindow is a knowledge mask, and a concept selector. A knowledge mask (or selector) allows the user to mask (or highlight) in reverse video some concepts or properties according to some criteria. Many criteria are available (Fig 5.18). Users can select parts of the hierarchy in a browser to be visible or invisible.

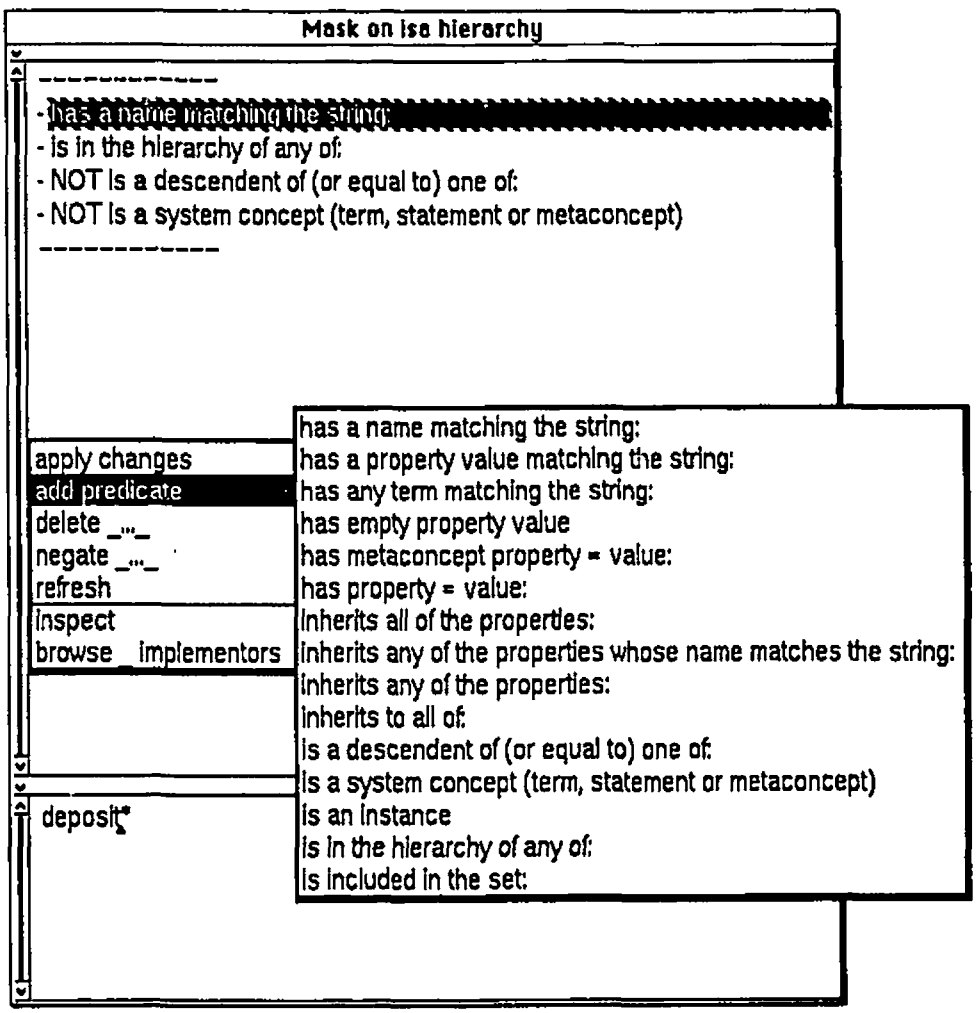

Fig 5.18 ATM Mask: shows a search for all concepts whose name do not start with 'deposit'

- A tabular or matrix form can present concepts and their important properties in two dimensions to permit the user to easily understand the similarities and differences between them:

  In general, tables help make decisions by focusing on the properties that specific concepts possess or do not possess. In CODE4, a matrix subwindow allows the editing of various kinds of inherently two-dimensional data; e.g. concepts may be displayed on one axis and properties on another. Cells in the matrix contain values of the statement involving a given concept and a given property. A Property Comparison Matrix (PCM) allows the user to view and compare properties for two or more concepts in tabular format (similar to a spreadsheet). These concepts may be siblings or arbitrarily chosen. The user can mask out concepts or properties or can have different options for viewing the matrix.

ATM Property Comparison: Transaction Methods

	deposit-anAmount	withdraw-anAmount	transfer-anAmount to:	get-theBalance
implementor	class Account	class Account	class Account	class Account
St protocol name	depositing	withdrawing	transferring	querying
comment	add anAmount to the balance	decrease anAmount from the balance	a transfer is a withdraw from the source Account then	extract the balance
returned expression	^ balance <FixedPointNumber>	^ balance <FixedPointNumber>	^ balance <FixedPointNumber>	^ balance <FixedPointNumber>
balance	balance is increased	balance is decreased	balance is depressed	n/a
self withdraw-anAmount	n/a	n/a	to decrease the balance of self	n/a
anAccount deposit-anAmount	n/a	n/a	to increase the balance of anAccount	n/a
Number -	n/a	to subtract a number from an another	n/a	n/a
Number <	n/a	to compare two numbers	n/a	n/a
Number +	to add two numbers together	n/a	n/a	n/a

Fig 5.19 ATM Property Comparison Matrix

- The ability to provide the user with overall control of the system: A control panel in CODE4 is a window used to configure the CODE session to the user's needs. The user can specify the level of expertise desired (as beginner, intermediate, expert, or developer). An environment mode is used to set parameters that apply to the system as a whole (including the font size and ClearTalk parsing). A Knowledge base control panel allows the user to manage different kinds of knowledge bases. Textual and graph format control panel are used to determine the appearance of browser subwindows using the outline and the graphical notation interaction paradigm respectively. A help control panel provides help about many aspects of the system.

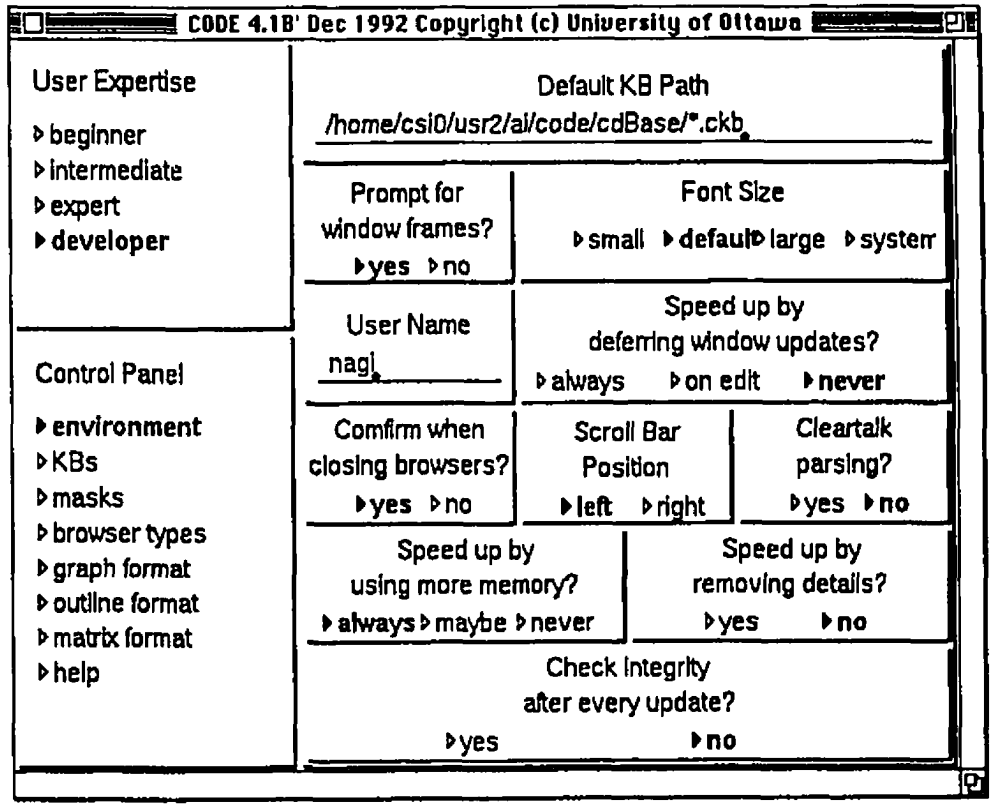

Fig 5.20 ATM Control Panel

User Interface Features Added to COSEE

- The ability to easily discover how a domain or design concept is reflected in the implementation i.e. how it is actually translated into a code:

  In COSEE, the user can dynamically link any of our three kinds of viewpoints (domain, design, or implementation knowledge) browser to the Smalltalk-80 browser if there is a need for more detail in understanding the coding or if there is a need to see the actual coding. Whenever the user changes a selection in any one of these open browsers, the other browsers change their selection; i.e. any browser is driven by other browsers. For example, in the ATM example, if the user changes the selection in the domain knowledge to the 'transaction concept' (or the 'printing property'), other browsers (design knowledge, implementation knowledge, and Smalltalk-80 browser) change their selection to the corresponding "Transaction" class (or the "printTransaction method").

- The ability to assist in "reverse engineering" existing code:

  In COSEE, the user can add the name of a class in the implementation knowledge viewpoint (or in the domain/design knowledge viewpoint through the use of pointers) and the system can automatically generate a subhierarchy corresponding to the one in Smalltalk-80, or the system can generate only one of its subclasses, as explained earlier in section 5.1.2.3. Also generated in the knowledge base with every new class are the class protocols, instance and class methods, and instance and class variables. Also, the system could generate, from the implementation knowledge, Smalltalk-80 classes and methods (with their variables); i.e. a step closer to automating programming.

5.3.3 Proposed Enhancements

In this section, we propose some useful features that could be added to both CODE and COSEE. We remind the reader that COSEE is an environment built on top of the knowledge management system CODE, for the sake of capturing the software development knowledge and creating a unified management system for software development.

5.3.3.1 Proposed Enhancements to CODE4

- The ability to automatically create a hierarchy (either concepts or properties) that corresponds in some way to an existing one with options to adapt it to the new location and meaning. Often the user wants to create a hierarchy that has a certain relation to an existing one (i.e. a function of it). For example, a hierarchy of implementation classes could be created corresponding to a hierarchy of design concepts. There should be a means for the user to map the names since we may want the same structure but with different names (for example, a rule that changes upper case to lower case or attaches a certain string to the names).

- The ability to perform additional operations on knowledge bases: For example, to split a knowledge base into two or more, merge two or more knowledge bases, copy and paste between knowledge bases (CODE's facility for this is only rudimentary).

- The addition of mechanisms that help extract knowledge from specific textual sources, consult a dictionary, and enter it as concepts into the knowledge base. Natural language documents are the major medium by which people organize and store knowledge; e.g. requirements specifications, or design descriptions. By scanning a document a sentence at a time, CODE could identify every noun and every verb phrase, get some of their properties

from a dictionary or from other knowledge bases, check them in the knowledge base, and consult the user before encoding them in the knowledge base (work has begun on this).

- The ability to access an on-line dictionary to give parts of speech or possible synonyms. This will help the software developers and maintainers to get quick definition of commonly used terms and concepts in the process of software development (work has begun on this).

- The ability to provide further assistance for structuring knowledge at the conceptual level. This will help the user in the knowledge acquisition task and also it will help in detecting conceptual contradictions and inconsistencies. For example, the system might suggest to the user what to do next and why. The system might also find problems with something the user has added to the knowledge base: For example, a concept that has conflicting properties (a certain introduced property contradicts an inherited one), or a property inherits two conflicting values from two different parents. The system could look at the values of a certain property at all its superproperties to check if it they have been changed in a consistent way. A pop up window critiquing the structure of the knowledge would provide a lot of help to the user in building a knowledge base.

- The ability to share knowledge bases concurrently (groupware). Thus, software developers can access and modify a shared knowledge base which they can all both add and retrieve from. The system should provide communication mechanisms, e.g. alerting anyone who could be affected by a change.

5.3.3.2 Proposed Enhancements to COSEE

The following mechanisms could be added to COSEE:

- A mechanism to dynamically link COSEE to a CASE tool (e.g. ObjecTime [Selic et al. 92]); that accesses the knowledge base, extract and incorporate the domain knowledge, and help end-users develop and maintain their own systems using high-level languages or diagramming tools.

- A mechanism to link COSEE to a documentation tool to have access to a complete documentation system, e.g. to export concept descriptions as near-English text.

- A mechanism to link COSEE to a formal specification system such as VDM [Jones 89]. This will help the validation of the specifications of the designed system. The benefit is that the designer will be able to locate and adapt specifications in the knowledge base, and then execute them. Test case data can be automatically generated.

- A highly restricted natural language mechanism to allow users to express requests in an informal-like and compact fashion, similar to the front end of LaSSIE.

- A mechanism to link COSEE to programming language environments other than Smalltalk-80 environment, like C++, Pascal, Lisp, and so on.

5.3.3.3 Enhancements to the Knowledge Base

The following knowledge bases could be available to COSEE to assist a developer further:

- A knowledge base that describes available software development tools in order to allow the system developer to select the most suitable and the most convenient tool.

- A software engineering ontology knowledge base that explains general concepts in the software development process: methodologies, tools, phases, people-involved, platforms, resources, heuristics, etc.....

- A knowledge base that describes all existing implementation modules that can be reused, give a particular choice of implementation language. For example, a knowledge base that explains all the Smalltalk-80 classes and methods could be very helpful to programmers (and especially novice ones). This kind of knowledge base should contain all knowledge needed by a developer except the actual code itself. The goal is to allow the programmer to easily understand a unit of code and then to reuse it.

Chapter 6

Summary & Conclusions

In this chapter, we will summarize what we believe our research has demonstrated and we will give some general conclusions about the relation of knowledge engineering to software engineering.

6.1 Conclusions from the Experiment

We believe our research shows a promising approach to providing a unified knowledge management environment for software development. We also believe that, when suitably developed and integrated with other tools, it could provide a better environment for software knowledge management than current non-integrated tools such as programming language environment, CASE tools, hypertext systems, or other knowledge-based software assistants. Only years of use in real software development can truly demonstrate this, however.

We addressed a major problem in developing a software system, i.e. the knowledge needed in every software engineering phase is scattered in different places and is not integrated. For example, the domain knowledge is often captured informally in a natural language document or formally for just the purpose of developing software systems and not for the documentation purpose. The design knowledge may be represented using a CASE tool that has no link to the tool used to capture the domain knowledge or even to

the implementation. The implementation knowledge may be buried in the code itself without any link to the tools used for capturing the domain or the design knowledge. The development environment does not integrate these different kinds of knowledge. The result is a group of separate systems and tools that render the software development process difficult and potentially "brittle". We have tried to provide a unified knowledge management environment for software development that can help eliminate the boundaries between all required software development knowledge and the development environment itself.

However, our environment cannot be a stand alone software development environment; it needs to be linked to various kinds of tools (such as CASE tools, formal specification systems, documentation tools) that will continue to be used as the primary means of software development for the foreseeable future. Although we acknowledge that there will certainly be some problems in the process of linking the development environment to other tools and systems (including a knowledge-based system), we believe that the contribution of this approach is worth further research. Most of these problems will occur if the development environment and these systems (or tools) are not implemented using the same programming language.

Our approach assumes that the software development process should begin with a natural description of the domain so that people may understand it without necessarily keeping in mind that a software system will be later developed. Thus, our main emphasis is to enable the software personnel to understand every concept in the domain knowledge. We have emphasized more the needs of the domain people, not the machine that implements the description; i.e. to describe the domain naturally rather than being based on certain design methodologies or programming languages.

Our conception for system development using COSEE is as follows:

- The analysis phase starts when the systems analyst encodes, with the assistance of domain experts, the domain knowledge in a knowledge base in CODE4. After a series of refinements to the knowledge base (collaborating with the domain expert), the systems analyst can start identifying concepts that might become good candidates for the design phase (a step toward reducing the gap between the analysis phase and the design phase).

- In the design phase, the system designer encodes the design knowledge in the knowledge base. System specifications are tagged for later validation and verification.

- In the implementation phase, the programmer simultaneously does the actual system coding and encodes knowledge about it in the knowledge base. Thus his/her knowledge is captured for those who will need it.

- The testing of the system can be done by testing every module in the coded system and verifying it against the design knowledge.

- The validation of the system can be done off-line by increasing the degree of formality in the knowledge base and by exporting it to an external formal specification system for checking. Such systems can reduce errors and, to a some extent, ensure that the designed system matches the requirements specification. For example, [Skuce and Mili 93] discuss the application of a formal specification system [Boudriga 92] to the ATM example. They focus on how to understand and validate the behaviour of objects given initially in terms of natural language descriptions of actions, or events, and sequences of events.

- The maintenance of the system would now involve knowledge management, done in all the three viewpoints of COSE. New

requirements and more domain analysis can be encoded in the domain knowledge by adding or replacing existing concepts. The rationale would be recorded as well. This modified knowledge might require some adjustment to the design knowledge and the implementation knowledge. The maintainer, if not familiar with the system, can understand the system from any or all these viewpoints and then start doing the maintenance in this knowledge base and in the actual system. If the maintainer performed code maintenance without updating the knowledge base, a mechanism in COSEE could be developed to detect the unmatched concepts resulting from comparing the implementation knowledge concepts and the actual implementation. It could automatically update some of the implementation knowledge as we have described. Compared to the Smalltalk environment as it currently exists, our environment would make available a considerable amount of new information in a highly organized and accessible structure. *The goal is to make this exactly the kind of information one seeks when trying to program in Smalltalk.*

We believe that COSEE can contribute to all these phases, in particular to the maintenance phase, since it is widely acknowledged that this phase consumes more than 70% of the software development cycle.

6.2 General Conclusions on the Relation of Knowledge Engineering to Software Engineering

Software engineering is an established discipline that has yielded more than two decades worth of tools and techniques. Knowledge-engineering, on the other hand, is an emerging discipline. Only recently have researchers tried to merge both disciplines. Representing knowledge about software is an important research area and a prerequisite to engineering expert-level systems to assist with software development.

Despite its lack of maturity, knowledge engineering promises to have a noticeable impact on software engineering in general. As Belady [Belady 91] points out: "For software engineering, until very recently a discipline unto itself requiring, basically, teams of people with CS degrees and an inclination to stay up nights with computers, is broadening, expanding in scope to intersect irrevocably with the discipline of knowledge engineering". He explains also that, as there is a growing demand to create large and complex software systems, there is a growing need for integration of applications, of hardware components, and ultimately of the people who use the system to work together across a network.

We believe that the future of software development will require the functionality of many systems collaborating smoothly to assist systems personnel in software development and maintenance. Thus, systems personnel can encode their knowledge (domain, design, and implementation) in a knowledge base using a knowledge management system, and can interact with a closely coupled CASE tool or other knowledge-dependent systems to access this knowledge base or other knowledge bases in distributed systems. As Chen et al. [Chen et al. 92] point out: "Tomorrow's complex, integrated applications will be developed using a combination of several enabling technologies (database- and knowledge-based systems, object-oriented technology, and hypermedia)".

Since software engineering is a knowledge intensive activity, addressing the software knowledge issue is a fundamental step towards solving the software crisis. We believe we have taken a small step toward freeing software development from some of its main problems.

Bibliography

Ambriola, V., P. Ciancarini, et al. (1991). "Towards Innovative Software Engineering Environments." *Journal Systems Software* 14: 17-29.

Basili,V. (1990). "Viewing Maintenance as Reuse-Oriented Software Development." *IEEE Software* January: 19-25.

Belady, L. (1991). "From Software Engineering to knowledge Engineering: The shape of the software industry in the 1990s." *International Journal of Software Engineering and Knowledge Engineering* 1(1): 1-8.

Bhansali, S. and M. Harandi (1990). "The role of derivational analogy in reusing program design". *Knowledge-Based Software Assistant '90*, Syracuse, NY, 28-41

Booch, G. (1991). "Object Oriented Design with Applications." Redwood City, CA, Benjamin/Cummings.

Boudriga, N., A. Mili, et al. (1992). "A Relational Model for the Specification of Data Types." *Computer Languages* 17(2): 101-131.

Chen, M. and R. J. Norman (1992). "A framework for Integrated CASE." *IEEE Software* March: 18-22.

Conklin, J. (1987). "Hypertext: An Introduction and Survey." *IEEE Computer* September: 17-41.

Conklin, J. and M. L. Begeman (1989). "gIBIS: A Tool for all Reasons." *Journal of the American Society for Information Science* May: 200-213.

Devanbu, P., R. J. Brachman, et al. (1991). "Lassie: A Knowledge-Based Software Information System." *Communications of the ACM* 34(5): 35-49.

Esp, D. G. (1991). "A Beginners experience of Smalltalk-80 for the evolutionary prototyping of an expert system". *Applications and Experience of Object-Oriented Design*, 5-10

Forte, G. and R. Norman (1992). "A Self-Assessment by the Software Engineering Community." *Communications of the ACM* 35(4): 28-32.

Freeman, P. (1987). "A Conceptual Analysis of the Dacro Approach to Constructing Software Systems."*IEEE Transactions on Software Engineering* 13(7): 830-844.

Green, C., Luckam, D., Balzer, R., Cheatham, T. and Rich, C. (1983). "Report on a Knowledge-Based Software Assistant." *Rome Air Development Center Report*: RADC-TR-83-195.

Greenspan, S. J. et al. (1988). "Toward an Object-Oriented Framework for Defining services in Future Intelligent Networks". *IEEE International Conference on Communications 88*, Philadephia, 867-873

Hayes-Roth, F. et al. (1991). "Frameworks for Developing Intelligent Systems." *IEEE Expert* June: 30-39.

Jarke, M. (1992). "Strategies for Integrating CASE Environments." *IEEE Software* March: 54-61.

Johnson, W. L., S. M. Feather, et al. (1991). "The KBSA Requirements/ Specification Facet: ARIES." *In Knowledge Based Software Engineering '91 Syracuse, NY* : 48-56.

Jones, C. (1989). "Systematic Software Development Using VDM (2nd edition)." London, Prentice Hall.

Kobsa, A. (1991). "Utilizing Knowledge: The Components of The SB-ONE Knowledge Representation Workbench." *Principles of Semantic Networks.* Los Angeles, Morgan Kaufman. 457-486.

Lenat, D. and R. Guha (1990). "Building Large Knowledge Based Systems." Reading, MA, Addison Wesley.

Lethbridge, T. C. (1991). "A model for informality in knowledge representation and acquisition." *Workshop on Informal Computing,,* Santa Cruz, Incremental Systems.

Lowry, R. (1991). "Software Engineering in the Twenty-First Century." *Automating Software Design.* Cambridge, MA: AAAI Press/MIT Press 627-654.

Lucarella, D. (1990). "A model for Hypertext-based information retrieval". *The First European Conference on Hypertext*, Paris, France, 81-94

MacGregor, R. (1991). "The Evolving Technology of Classification-based Knowledge Representation Systems". *Principles of Semantic Nets.* San Mateo, CA, Morgan Kaufmann. 385-400.

Majidi, M. and D. Redmiles (1991). "A Knowledge-Based Interface to Promote Software Understanding." *CA: IEEE Computer Society* : 178-185.

Nash, C. and W. Haebish (1991). "An Accidental translator from Smalltalk to ANSI C." *OOPS messenger* 2(3): 12-23.

Neighbors, G. (1984). "The Draco Approach to Constructing Software from Reusable Components." *IEEE Transactions on Software Engineering* 10(5).

Nerson, J.-M. (1992). "Applying Object-Oriented Analysis and Design." *Communications of the ACM* 35(9): 63-74.

Nielsen, J. (1990). "The Art of Navigation Through Hypertext." *Communications of the ACM* 33(3): 298-309.

Norman, R. J. and M. Chen (1992). "Working Together to integrate CASE." *IEEE Software* March: 12-16.

Norman, R. J., W. Stevens, et al. (1991). "CASE at the start of the 1990's". *4th International Workshop on CASE*, Irvine, CA, 128-139

Pugh, J. and W. Lalonde (1990). "Inside Smalltalk." N.J., Prentice Hall.

Ramesh, B. and V. Dhar (1992). "Design Rationale Capture and Use in Remap Project". *AAAI workshop on Design Rationale capture and Use*, San Jose, 221-225

Reubenstein, H. B. and R. C. Waters (1991). "The Requirements Apprentice: Automated Assistance for Requirements Acquisition." *IEEE Transactions on Software Engineering* 17(3): 226-240.

Rich, C. and R. Waters (1989). "The Programmer's Apprentice: A Research Overview." *IEEE Computer* 21(11): 10-25.

Robson, D. J. a. B., K. H. and Cornelius, B. J. and Munro, M. (1991). "Approaches to program Comprehension." *Journal Sys. Software* 14: 79-84.

Rumbaugh, J., M. Blaha, et al. (1991). "Object-oriented Modeling and Design." Englewood Cliffs, NJ, Prentice Hall.

Sametinger, J. and G. Pomberger (1992). "A hypertext system for literate C++ programming." *Journal of Object-Oriented Programming (JOOP)* January: 24-29.

Schoen, E. a. S., R. G. and Buchanan, B. G. (1988). "Design of Knowledge-Based Systems with a Knowledge-Based Assistant." *IEEE Trans. on SE* 14(12): 1771-1790.

Selfridge, P. G. (1990). "Integrating Code Knowledge with a Software Information System". *Knowledge-Based Software Assistant '90*, Liverpool NY, 183-195

Selic, B., G. Gullekson, et al. (1992). "ROOM: An Object-Oriented Methodology for Developing Real-Time Systems." *CASE'92 Fifth International Workshop on CASE.* Montreal, Quebec, Canada.

Skuce, D. (1992). "A Review of 'Building Large Knowledge Based Systems'' by D. Lenat and R. Guha." *The AI Journal* to appear.

Skuce, D. and A. Mili(1993). "Behavioral Specifications in Object-Oriented programming". in preparation.

Skuce, D. and T. Lethbridge (1992). "A Knowledge Representation for Interactive Knowledge Management." in preparation.

Smeaton, A. F. (1991). "Retrieving Information from hypertext: issues and problems." *European Journal of Information Systems* 1(4): 239-247.

Smith, D. R. (1990). "KIDS: A Semiautomatic Program Development System." *IEEE Transactions on Software Engineering* 16(9): 1024-1043.

Tan, Y. M. (1989). "A Proposed Program Design Apprentice." *International Journal Conference on Artificial Intelligence '89.* Detroit, MI, 272-278.

Urban, J. E. and S. Kaphan (1992). "The impact of Undergraduate Software Engineering Education on CASE Tools." *Int. Journal of Software Engineering and Knowledge Engineering* 2(2): 263-276.

Waters, C. Richard (1981). "The Programmer's Apprentice: A Session with KBEmacs." *IEEE Transactions on Software Engineering* 11(11): 1296-1320.

White, J. L. and S. Kaphan (1989). "Implementing Lisp on standard hardware." *Sun Technology* 2: 63-70.

Wirfs-Brock, R., B. Wilkerson, et al. (1990). "Designing Object-oriented Software." Englewood Cliffs, NJ, Prentice Hall.