Knowledge management in software design: a tool and a trial

by Douglas Skuce

The paper describes experiences in applying a knowledge management tool, called CODE, to the major knowledge management problems in the design of a large commercial software system. There is explanation of how the design of such a product was documented as a CODE knowledge base by using CODE to carefully analyse all the important concepts and terms in direct consultation with the developers. CODE could be described variously as a 'design knowledge capture tool', a 'conceptual design assistant', or a 'knowledge management system for technical documentation'. Its knowledge base and its highly interactive interface combine ideas from frame-based knowledge representation systems, object-oriented systems and hypertext systems. CODE is intended to augment and eventually replace conventional word-processing and graphical tools for the capture, editing and documentation of such knowledge. Hence it facilitates constructing definitions or descriptions of all the main concepts in a system, validating these so that the system designers are in agreement, and retrieving and graphically displaying such knowledge in various formats. CODE has been used in a wide variety of knowledge management situations, but here the focus is on its use in software engineering design.

1 Introduction

The need for tools and techniques to enhance software productivity is indisputable [1]. Hence interest is rapidly growing in knowledge-based software engineering which employ techniques from knowledge engineering and AI. The conferences on Knowledge Based Software Assistants [2] illustrate this trend. We include under this term any useful technique for representing knowledge beyond conventional documentation techniques, for example, hypertext. It is increasingly being recognised that many software development problems are the result of misunderstanding and imperfect communication between software production team members, including misunderstanding of the requirements or misunderstanding of components being reused. Knowledge-based approaches can assist in capturing, debugging and the disseminating the type of knowledge normally consigned to natural language documents, which are often not adequately clear, detailed, easy to find, or consistent.

The research reported in this paper represents a confluence of two research areas: software engineering and knowledge management*. We describe a knowledge management tool called CODE (Conceptually Oriented Design Environment) [3] and explain how CODE was applied to acquiring, debugging, and documenting design knowledge for a large commercial software system under development. CODE's object-oriented knowledge base and its highly graphic interface permitted experimentation with definitions or descriptions of all the main concepts in the system under development. By interacting with CODE, the system designers were able to come into better conceptual agreement, i.e. they shared a more uniform conception of the system, and terminological agreement, i.e. they clarified what to call these components, a common cause of confusion. The resulting knowledge base served as a semi-formal specification that anyone could access for definitive and unambiguous answers to many high-level design questions. The main users of the knowledge base were the system designers, the documentation team who had to prepare the various types of manuals, and maintenance programmers who needed to understand the structure and rationale of the system. Everyone who needed knowledge about the system benefited, because the knowledge base served as a unifying 'encyclopaedia' of the whole system.

* This term is now coming into common use. It refers to all operations required on knowledge, such as initial acquisition, clarification and debugging, formal specification, automatic processing (e.g. checking), retrieving, converting to other forms, document production etc.
The most commonly used means of communication in software development is natural language text augmented by diagrams and possibly some formal notation. Such information is usually stored in various formats processed by incompatible tools, e.g. word processors, 'draw' programs or various CASE tools. By using these uncoordinated and incomplete techniques, which provide virtually no assistance with language, understanding and standardisation of concepts, terms and the details of a design can be unclear, leading to costly confusion and errors.

A real example of this problem is difficult to provide explicitly within the confines of a paper, but few would question that software engineers often experience such problems. Later, when we discuss our trial application of CODE, we describe several problems. Here, we list some of the kinds of knowledge management problems often encountered in practice:

- A document is not up to date, is incomplete, or contains undefined terms.
- A glossary is incomplete and the definitions are vague; the uses of a term may not conform to its definition; it may not be clear that a term has several meanings, even in one document.
- A diagram or table that accompanies a document is not properly co-ordinated with it, or does not exist, when one would help.
- Staff keep their own idiosyncratic notes on their activities, and others either cannot access them or cannot understand them.
- There is no one authoritative source for all knowledge about a project; knowledge tends to be distributed in various incompatible media, and no one knows where everything is.
- Meetings are often held too informally, without recording decisions, questions, plans etc.; knowledge is lost from white boards; reasons for decisions (design rationale) are unrecorded.

Our goal then is to demonstrate a unified approach to such knowledge management problems.

We first review related research and briefly introduce the basic ideas of CODE. The actual software components of the CODE system are described, and although the details of CODE have been described previously [3], we have included some of these here so that this paper is complete on its own. We also describe an experiment or trial in which CODE was used to capture a description of a large commercial software system, and offer some suggestions for future development and conclusions.

1.1 Related research

Before explaining our approach to knowledge management problems, we briefly describe some related research. Different approaches focus on different kinds of knowledge. The kind of knowledge we are concerned with is 'conceptual' and 'semi-formal' knowledge, i.e. abstracted knowledge about the concepts and ideas represented in the software, expressed with a level of formality between that of arbitrary natural language and a mathematically based formalism such as VDM. What makes this knowledge particularly difficult to capture is that it relies heavily on the meaning of terms, many of which are far from standardised and often cannot be defined formally at all. Thus, our work emphasises concept description and definition on a semi-formal level.

There have been general discussions of AI in software engineering previously [4], including discussing general aspects of knowledge-based software engineering tools. Our research focuses on higher level or conceptual knowledge, rather than on the code details, and we only cite similar research. This emphasis on concepts rather than code is reflected in an increasing concern for requirements analysis, which is mostly informal. For example, the group at MIT which has long worked on AI in software engineering has shifted their emphasis from the code level to requirements, which we include under the conceptual level [5]. Requirements knowledge tends to depend heavily on terminology, but few papers in the computer literature specifically point out the importance of terminology [6].

Some systems emphasise the co-operative work aspect and often are hypertext-based. Cybulski and Reed [7] describe a hypertext system for general software knowledge capture, and Sobiesiak and Mylopoulos [8] discuss a system whose purpose is similar to the system discussed in this paper: a hybrid of knowledge-based and hypertext techniques for multiple authoring of documents. Recently, a workshop has been devoted to the problem of capturing 'design rationale' [9], although this workshop dealt with all types of design, not just software, and features mainly hypertext approaches, e.g. glibis [10] which is perhaps the best known of these. These hypertext-oriented projects reflect the trend to capturing semi-formally the kind of knowledge that previously was only expressed in natural language documents. However, hypertext systems lack an AI component to provide the inferencing that we deem essential.

The work by Devanbu et al. [14] is somewhat more related to CODE: it describes the use of an AI knowledge representation language, Kandor, to capture knowledge about a commercial software system, a somewhat similar situation to our trial. However, the system/language and situation differ from ours in many ways:

(a) Kandor was conceived as a formal knowledge representation language with certain strict logical properties.
(b) The Kandor system was not conceived as a knowledge acquisition or management system.
(c) There is no discussion of how such a tool might be integrated into the overall software process.
(d) There is essentially only one mechanism by which the system detects errors, i.e. classification.
(e) There is no description of the methodology for building the knowledge base.
(f) Devanbu et al. focused on knowledge about actions, whereas we have focused on knowledge about the entities comprising the system.

Other requirements-oriented systems with some similarity in purpose or functionality to ours include the ARIES system [15], which provides a graphics environment for
transforming informal requirements into formal specifications, and the work of Palmer and Fields [16].

Many of the systems described in the literature on knowledge acquisition are generic, i.e. they are applicable to any application domain, but often are oriented towards the development of rule-based systems. Unfortunately, software engineering has not been a prominent application of knowledge acquisition. The field is typified by the Knowledge Acquisition for Knowledge-based Systems Workshop conference series [17] and the International Journal of Human-Computer Studies. They often include other topics besides just the acquisition process, and hence we prefer the more general term 'knowledge management'.

Many well known knowledge acquisition systems deal only with highly constrained types of knowledge, such as so-called 'object attribute-value' triples or even just 'object attribute' pairs (without a value). A review of some well known systems of this type can be found elsewhere [18]. This limited degree of expressiveness or the inability to perform inferencing at all hampers many of these systems. We have found that the type of knowledge needed in software description requires a considerable degree of expressiveness and at least an inheritance-based inference mechanism, and it must permit a range of formality; the user must be able to process both expressions with formal syntax and semantics and natural language phrases. Other general-purpose knowledge acquisition systems with frame-based knowledge representations include those of Motta et al. and Koelsch [19, 20]. However, many of these systems lack features that our experiences have caused us to add to CODE, such as support for terminology and simple natural language syntax, database-like retrieval functions, the ability to draw various kinds of diagrams, support for group interaction, operations like finding all occurrences of a phrase and renaming, and a flexible degree of formality.

Generally speaking, in the above-cited research, some knowledge representation technique, usually object-oriented ('frames' in AI parlance), is used as an environment that permits at least entering and editing knowledge, but often with insufficient emphasis on debugging, displaying or retrieving knowledge, and rarely with support for linguistic issues. Thus, these systems illustrate the kind of deficiencies that we have tried to alleviate in our research.

2 Short description of CODE

The version of CODE (version 2) used in the present application has been described in detail previously [3]. Thus, we describe the CODE version 2 system only briefly: its motivation, basic concepts, i.e. its underlying knowledge structuring categories or ontology, and its knowledge representation.

CODE is a general-purpose knowledge management system for knowledge processing tasks such as acquisition, debugging, retrieval or reformattting. CODE has been used in a variety of other applications besides software engineering, such as terminology [21]. The design of CODE is based on earlier experiences in capturing software knowledge, where we developed an advisor for a commercial fourth-generation database query tool using the common knowledge engineering tools of the mid-1980s such as KEE† and ART†. We sought to capture semi-formally the main design concepts, structures, rationale and the associated terminology of a large software system under design. By semi-formally we mean with a precision considerably beyond that of natural language documents but stopping short of very formal description techniques. We feel that the former often do not describe the conceptual structure of a design sufficiently precisely, nor do they properly define and control its terminology. On the other hand, we consider that the formal descriptions are too rigid, limited and abstruse to be appropriate in all but a few very demanding applications. They rely totally on mathematical concepts, ignoring the problem of terminology that must also be addressed as there must still be accompanying documents. Usually only parts of a system are described formally.

Therefore, we have sought a knowledge format beyond that of a structured document (e.g. hypertext) but one that is not too formal. In CODE, you may enter only unstructured English-like phrases that are not checked at all, you may use certain conventions on syntax, termed ClearTalk (described in Section 3.2), which can be checked by CODE for syntax, terminology and some logical properties, or you may enter expressions in a formal language. Any checking of such formal expressions is beyond the CODE domain. However, CODE can serve as an organiser and repository for such expressions, and as a bridge between the informal and the formal.

In our trial application, we followed a middle ground. The human operators did what they are best at, spotting problems, while CODE quickly retrieved and displayed clearly all the knowledge needed in a simple unified format so that potential problems were highlighted. To achieve automatic detection of all errors would require an extreme degree of formalisation and logical functionality, which we feel is beyond what most software developers are willing to do or need. There is also the problem of finding errors in the formalisation, to say nothing of computational difficulties. Nearly all the problems we encountered in our knowledge involved a deep understanding of the terms and concepts, and we do not believe these could have been detected automatically. Thus, we eschew totally formal methods.

To summarise, here are the three main functions CODE provided:

- to act as a medium for knowledge capture, transfer, and iteration, by which a team of designers were able to document the essential concepts and terminology in a manner (they judged) superior to conventional documents, the media they would have otherwise used;
- to assist in debugging the knowledge. Users could easily spot and correct many conceptual, logical and terminological errors. Thus, the designers were able to identify disagreements and then come to agreement on shared conceptual structure and terminology.
- to serve as a knowledge source, for those who need rapid access to easy-to-understand descriptions and/or

† Trademarks.
‡ This is not to say that conventional documents were not used. As we see later, the CODE kb served as a resource from which such documentation was developed.
specifications related to software development. In our application, such people included the top-level designers (specifications), the implementors, maintenance personnel (system documentation) and end users (user documentation).

2.1 CODE ontology

The ontology (basic semantic categories) of CODE is organised around the notions of concepts, the ideas or topics of interest, and their properties. Usually in CODE we describe or define mainly class (or type) concepts, which may have instances (examples), which are also concepts and are treated almost identically. A type specifies properties, either completely or partially, that its instances will possess. For example, the type (class) 'actor specification' has as instances actual specifications of actors, the fundamental building blocks (the main 'objects') in the Telos system (see Section 4). Knowledge is organised into units termed conceptual descriptors (cds), which are analogous to frames in AI or objects in OOP (object-oriented programming).

Fig. 1 shows a typical view of a CODE screen. In the lower left part of the Figure, the cd for 'actor specification' is partly visible. We can also see where this concept fits in the main picture in the inheritance hierarchy graph underneath, and in the browsing window on the right. (These ui components are described further in Section 3.1.)

A cd should be thought of as a packet of statements describing the properties of some concept termed the subject of the cd. A large cd would be analogous to a series of paragraphs or document section.

A property corresponds to a short natural language statement with one verb and is the main unit of information that may be added, removed, modified, or inherited to subconcepts (subclasses or instances). Properties are divided into a fixed set of system properties and an open-ended set of user properties. The former control basic operations like naming cds and the inheritance structure. Properties usually have values. The expressions naming properties or used as values in these examples are in ClearTalk, a controlled English-like syntax that is optional.

User properties are divided into various categories to assist the user in structuring knowledge. The following are example categories:

attributes provide information about the subject of the cd alone and do not refer to anything else. For example, we might have in a cd for 'person':

sex: one of 'male, female'

The expression following the colon is the value (here in ClearTalk), which refers to a class or attribute value in the case of attributes.

related concepts provide information about other concepts related to the subject as 'peers', i.e. they are usually subjects of other cds. An example would be the father of a person:

father: a man

§ Here we use some everyday examples to avoid misunderstanding due to a lack of familiarity with the concepts.
cd name: actor specification

super concepts: class entity specification, user defined specification

comment (o h v) (T) & (actor specification): # describes a class of actors. Users can define actor specifications in class hierarchies.

concept owner (n v m) (T) & (actor specification): CL.

level (r n v) (T) & (actor specification): 4.

status (n h v) (T) & (actor specification): accepted.

ATTRIBUTES WITH VALUES

class name (l r o) (class entity specification) & (actor specification): an actor class name.

class name space (c r n) (class entity specification) & (actor specification): a set of actor names.

designate date (c r o) (user defined specification) & (user defined specification): a date.

visual representation (l r o) (entity) & (specification): some of text, iconic.

COMPONENTS

behaviour ports (l r o) (actor specification) & (actor specification): a set of end port references

bindings (l r o) (actor specification) & (actor specification): a set of bindings specifications.

component actors (l r o) (actor specification) & (actor specification): a set of actor references.

deep structure (l r o) (actor specification) & (actor specification): the union of bindings, the set of specifications of the component actors, behaviour ports, and external ports; these components of a # and the deep structure of these external ports (l r o) (actor specification) & (actor specification): a set of relay port references.

CONSTRAINTS

compatibility (l r n) (actor specification) & (actor specification): an # X is compatible with an # Y if X is a compatible descendant of Y or Y is a compatible descendant of X.

name space containment (l r n) (class entity specification) & (class entity specification): the class name is a member of the class name space.

EVENTS ON

create (l r n) (entity) & (user defined specification): # is made to come into existence at some point in time, by a user.

destroy (l r n) (entity) & (specification): # is made to cease to exist at some point in time by some agent: # is removed from the historical record of specifications.

incarnate (l r o) (actor specification) & (actor specification): an incarnation of # is created as a top level actor incarnation or a reference to # is incarnated.

RELATED ENTITIES

ancestor (l r o) (class entity specification) & (actor specification): an actor specification; that # inherits properties (usually components) from; corresponds to superclass.

compatible descendant (l r o) (actor specification) & (actor specification): = can be substituted with.

descendants (l r o) (class entity specification) & (actor specification): a set of actor specifications; corresponds to subclass.

incarnations of # (l r o) (specification) & (actor specification): a set of instances of actor incarnations.

inherited properties (l r o) (class entity specification) & (class entity specification): a set of property bodies; the items # inherits from its ancestor.

references to # (l r o) (class entity specification) & (actor specification): a set of actor references.

supercomponents (l m) (entity) & (actor specification): a specification.

Fig. 2 Detailed listing of a concept descriptor for actor specification

---

Fig. 3 Property graph showing relationships between the concepts 'port reference on an actor reference', 'actor specification' and 'static actor reference'
An important subcategory is components, i.e. parts either in a physical or functional sense:

- **arms**: two human arms

**constraints**: the CODE system can be linked to an external first-order logical deduction system FOLDE, written in Prolog, which permits certain logical inferences to be performed. For example, in the cd 'adult' we might have the property 'age' and the following constraints:

- **lower age limit**: 17 < age
- **parent age constraint**: if X is a parent of this person, then age of this person < age of X

FOLDE can do forward or backward inference on such constraints if desired. For example, if a subconcept of 'adult' was defined with an age of less than ten, this would be detected.

**operations** (also termed events): these specify actions that may be applied to or performed by the subject concept. Part of understanding something is understanding its behaviour, i.e. events. For example, in the Telos system for the concept 'class browser' we might have an event called

- **open**: open # on menu choice

The subject concept # ('class browser') will be implemented as an actual class in Telos. Implementation or code level properties such as these could be automatically generated from the host language (in this case Smalltalk) or vice versa, a prospect we are currently investigating. Thus, CODE could become a sophisticated browser allowing exploration of actual implementation details in addition to the conceptual level details. These ideas have been explored further [23].

**CODE** uses a frame-like inheritance mechanism, but with a very comprehensive notion of property (slot) beyond that found in most frame-based systems. In the interests of brevity, we do not describe these mechanisms in detail.

### 3 CODE system components

The CODE system is written in Smalltalk 80 and Prolog, and runs on UNIX platforms, Macintoshes and 486s, occupying about 2 Mb (excluding the Smalltalk system itself). The user interface consists of three main components: the cd views, the grapher and the property browser. Fig. 1 shows an image of a complete SUN screen display on which some of these features can be seen. The two subsystems used for syntactic and semantic checking, ClearTalk and FOLDE, are written in Prolog and can be invoked from the interface in the UNIX version.

#### 3.1 User interface components

**3.1.1 Cd views**: the most informative interface component is the cd view, which shows the knowledge contained in all parts of a cd. The user can control what is displayed and how. An example of a cd view ('actor specification') can be seen in Fig. 2.

A cd view is always maintained up to date with respect to the whole inheritance network. If potential conflicts are detected during property inheritance (e.g. two properties of the same name inheriting from different sources), the user is prompted to resolve the conflict. Hypertext-like access is also available; we may highlight a phrase and see if the system can find out anything about it, e.g. is it a concept, and if so, jump to its cd.

**3.1.2 Grapher**: most interactive knowledge acquisition systems incorporate some kind of graphical assistance to aid in visualising at least the inheritance network or other graph structures. Our experience confirms that this is an essential feature, and the graph features of CODE are highly developed. It is possible to open one or more graphical views of the network. Portions of the graph may be selected for moving, deleting, writing out the knowledge base as ASCII files, enlarging in another view, temporarily hiding etc. Individual cds may be selected for opening (making their cd view visible on the screen), renaming, deleting, writing out etc. Graphs showing property relationships ('semantic nets') may be drawn. The main hierarchy graph can be seen as one of the (partially hidden) views in Fig. 1, and Fig. 3 shows a property graph constructed by selecting several concepts on the main graph for which it was desired to see the interrelationships.

**3.1.3 Property browser**: browsers allow very convenient locating and editing of system components, in particular making many rapid menu selections or locating relevant information quickly. To add or modify any property of any cd, the user opens a property browser, shown in Fig. 4. The user sees a list of concepts (i.e. cds for these), with the properties of the selected concept. Any of the facets (parts) of the property may be edited, and any
change is immediately inherited to all subconcepts and appears updated in any cd views that are on the screen.

3.2 Subsystems: ClearTalk and FOLDE

CODE offers optional support for natural language, particularly in controlling syntax and terminology, centered around two features: ClearTalk, a simple English-like syntax that can be easily learned by users and checked by CODE; and a lexicon system that permits the accumulation and monitoring of terminology. The use of ClearTalk coupled with a controlled lexicon can greatly reduce the possibility of misunderstanding due to complex or ambiguous sentence constructions, or inconsistent or undefined terminology. Both these features were heavily used and judged important in the trial. ClearTalk constrains the syntax and semantics of simple noun and verb phrases and whole sentences. The syntax and semantic interpretation of ClearTalk cannot be fully described in this paper [24].

A logical subsystem called FOLDE is associated with CODE. It is a full first-order logical deduction system specialised in debugging sets of rules and facts, written in Prolog. We do not discuss FOLDE further here in the interests of brevity and because it was not used in the trial, for reasons described below.

4 Capturing software knowledge: a trial

In this Section, we describe a trial in which CODE was used to capture a significant amount of knowledge about a large commercial software system. The Advanced Systems Design group of Bell-Northern Research (BNR) was developing a design workbench called Telos, a highly graphic object-oriented system for designing large distributed real-time systems [25]. At the time of the trial, Telos had taken tens of working years of development and was reaching the 'alpha' test stage. However, at that time, the existing documentation was sparse.

Most of the knowledge about the design still resided in the designers' minds. Therefore, CODE was used to create descriptions of the basic concepts in Telos using about 300 cds. Each cd had, on average, about 10–15 properties; some of them inherited without change, some inherited with changes to the value and some newly introduced. To give some idea of the size of the knowledge base, it would be the equivalent of a document containing several thousand sentences, where each property would correspond to a short sentence. When printed out in 'raw' form, it was about 200 pages.

The task of acquiring and debugging the knowledge base took roughly 200 working days, although perhaps 20% of this was devoted to discussing knowledge engineering and CODE concepts in general. This represented, however, only a few percent of the project's human resources. Had CODE not been available, some (but probably not all) of this knowledge would have been recorded in the traditional manner; rough documents written by the experts would have been refined into better ones by documenters, who would have had to spend weeks (typically not enough) asking the experts for clarifications and additions. Another advantage was that users could access knowledge as soon as it was entered into CODE, rather than wait for it to appear in documents.

To actually prove that the use of CODE was a definite advantage, however, would have been a prohibitively expensive experiment; both conventional and CODE-based knowledge capturing methods would have to be tried in parallel and then compared in some objective way. The trial was expensive as it was, and there was considerable debate initially as to its worthiness. Thus we cannot think of a way to perform such a trial in a more objective manner that would not severely burden normal commercial budgets and timetables; indeed, we know of no such experiment reported as yet.

The resulting knowledge base provided a highly structured description of the conceptual structure, terminology, properties, constraints, assumptions etc. of the system. The main purposes it served were the following.

- It assured the designers that they were in agreement on concepts and terminology. Judging by the time it took to resolve many such discrepancies, lack of agreement represents a major bottleneck that is often unrecognized; it manifests itself in unclear communication and design errors caused by misunderstanding that are normally hard to pin down. Often it becomes explicitly apparent only during documentation, a phase that is too often left too late, or worse, as bugs in the system.

- It helped resolve problems in the proposed design of a subsystem by clearly stating a proposed set of concepts and their properties, i.e. a requirement. By using CODE it became easier to see (i.e. than with less structured methods) what the problems were and how they affected the rest of the system.

- It provided a unified standardised source of knowledge about the system. The main groups needing this knowledge were the system designers, implementors and the documenters. Although the designers and implementors gained a better understanding of some parts of the system by interacting via CODE, the prime benefactors in fact were the documenters, as much of the knowledge that they required was already highly refined in CODE. Of course, anything that improves how the documenters do their job benefits everyone, for the documentation improves and the demand on the designers to supply knowledge to the documenters is reduced. The key point is to capture the knowledge as completely, succinctly and as early as possible.

- Although still not complete nor totally validated, the designers believed that the knowledge base nevertheless far exceeded in completeness, correctness and consistency any natural language descriptions of Telos that would have been produced by the normal means of documenting the design. Some of the time invested in building the knowledge base was recovered by shortening the time that documenters needed to understand the system and its terminology, although it would be difficult to quantify this. To further assist the documenters, CODE can print out cds as a fairly legible document if the user follows ClearTalk conventions (Fig. 2 shows an example of an actual cd listing derived directly from the knowledge base, and Fig. 5 shows the pseudo natural language output).

† This system, renamed ObjecTime, is now marketed by ObjecTime Ltd, Kanata Ontario.
A binding specification is a non-class specification and a user-defined specification. It is an abstraction for a connection between two actors, achieved through ports, and supported by the communication service. An instance of a binding specification is THE binding spec of A.

A binding specification can have the following ATTRIBUTES WITH VALUES:
- a visual representation, which is iconic
- a designer name, which is a person's name
- a design date, which is a date

A binding specification can have the following RELATED ENTITIES:
- It is a component of an actor specification that a binding specification is part of
- an incarnation of a binding specification, which is a subclass of binding incarnation
- ports, which are a set of two port references.

Fig. 5 Example of documentation produced directly from CODE's graphic and pseudo natural language output (the sentence beginning 'It is ...' was simply an unprocessed comment.)

These were imported directly into publishing software. Such printouts were also useful as a means of easily communicating the current state of knowledge about specific topics without having to log into CODE itself. Designers found it convenient to attend knowledge acquisition or debugging sessions after noting points of disagreement or uncertainty on such printouts, i.e. they acted like an agenda.

4.1 Methodology: using CODE

Briefly, our methodology for building the knowledge base was as follows. One at a time the four main designers of Telos would join a knowledge engineer (often the authors) in intensive sessions lasting about two hours, where the designer would describe concepts and properties of some part of the system. Each designer was assigned as prime to those concepts with which they were most familiar. The process would usually involve much discussion and scribbling of rough diagrams on paper, as the designer attempted to explain the concepts to the knowledge engineer and clarify their own thinking. When the knowledge engineer (KE) thought the designer (D) had settled on a statement, they or the designer would enter it into CODE and ask the designer to criticise it. Exchanges between them would include interactions such as*

D: Actor normally has a set of incarnations associated with it at run time.
KE: Did you mean each actor specification? (the term 'actor' is ambiguous)
D: Yes (so KE opens actor specification cd)
KE: Is the association only at run time? (something makes KE doubt this)
D: Well, at design time, there is something similar but these things aren't the same. They are like parts of an actor specification.
KE: What are they? Let's look at the currently known parts; (checks knowledge base; makes an educated guess) Are they the actor references?
D: Maybe they are. I'm not sure, because we only started using this term a few days ago, and I'm not yet clear if it is the same as the old thing we used to call usages'.
KE: OK, let's look at the current cd for 'references'. I remember when Joe and I decided to rename it, he also changed some of the properties.
D: Well, it looks mostly like the old 'usages' thing, but can you show me what he changed?
KE: (applies mask to show only changes in the cd made by Joe since last week) There.
D: Ah-ha! Now I see! He has decided to allow usages, I'm sorry, references, to be optional parts of actor specifications. (usages, I mean, references, used to be necessary, i wonder why he no longer considers them necessary? This would mean we would have 'empty' actors! Maybe more than the name has changed. I wonder what he does with leaf actors? I don't think they can be empty. We had better get this straightened up right away, because my staff are coding this now.

Typically once a week there would be 'group reviews', in which the concepts not yet clarified were discussed in a meeting of all four designers using CODE on a Sun workstation. Indesirable consequences could easily be spotted by suitable queries using CODE's retrieval facility, e.g.

'show all concepts added by John since last week.'
'show the every concept with the property 'end ports'.'
'show all occurrences of specifications that refer to actors'.
'show all occurrences of the term 'actor reference'.
'show all the things that can be sent a message by an actor.'
'show how the property 'components' changes down the 'controller' hierarchy.'

The resulting displays were sufficiently detailed and restricted that the designers could quickly examine just the knowledge they wanted and could easily spot problems. Again, this reflects the major design philosophy of CODE; to organise and display knowledge sufficiently clearly so that humans can usually spot the problems. The

* Readers should not conclude from this concocted dialogue that this degree of uncertainty was typical. Such situations do occur, however, and hence our desire to alleviate them.
concept hierarchy diagram was the single most useful graphical aid and, like all others, it is drawn automatically from the knowledge base.

Several iterations were usually needed to reach consensus on many concepts and properties, even some previously believed to be well understood and agreed upon. Referring again to Fig. 1, in the browser on the right, the portion of the knowledge base above and below specification has been shown, and actor specification selected. Here is a brief description of a few of its property categories and their properties:

- attributes with values
  - class name: an actor class name
- components
  - behaviour ports: a set of end port references
- constraints
  - actor spec cl: the external ports are disjoint from the behaviour ports

A ClearTalk output of this knowledge with English explanations (in italics) follows:

Each actor specification has some attributes with values, e.g., a class name, which is an actor class name. Actor specifications are classes, and each class must have an actor class name.

Each actor specification has some components, some of which are called behaviour ports. These are a set of end port references. Actor specifications have various parts or components. These components include behaviour ports. They must be end port references, i.e., to specify the behaviour ports, you must put end port references in an actor specification.

Each actor specification has some constraints, e.g., actor spec cl, that the external ports are disjoint from the behaviour ports. This is a formal way of saying that behaviour ports cannot be external, and vice versa.

4.2 Conclusions from the trial

The following general observations and conclusions summarise the opinions of most of the participants, drawn from an informal questionnaire.

- Someone trained in knowledge engineering is absolutely essential to oversee the knowledge acquisition process. A considerable amount of experience is required to resolve the variety of problems encountered, e.g., on matters such as how to structure the knowledge, how to choose terminology, how to express logical constraints etc. After potential knowledge engineers have had some training in conceptual analysis and one or two months of exposure to CODE, the preferred mode of use would probably be to have them enter their ideas into the system first in a rough draft or brainstorming mode. Subsequently, we would permit group interactions in which the other engineers criticise the knowledge content, refine it, add to it etc. In the future, such group interaction should be interactive and machine-assisted in a 'groupware' fashion.

- Had the CODE system been available from the outset of the Telos system design, many of the problems we experienced in trying to clarify the conceptual structure and to discard entrenched but undesirable terminological choices might have been avoided.

- ClearTalk, i.e., the restricted use of terminology with a simple syntax, was seen as a definite advantage and was widely used. At first, there was hesitation, as people tended to want to express themselves in unconstrained English. As the knowledge engineer repeatedly pointed out ambiguities or outright errors and suggested how the same thing could be said in ClearTalk, its acceptance increased so that the designers began trying to express themselves directly in ClearTalk, although they tended to rely on the KE to assist them.

- No use was made of the FOLDE engine, although some logical rules were written that were sufficiently simple that FOLDE was not needed. This was a surprise to some because we might anticipate many opportunities to make precise logical formulations of design constraints. Indeed, the designers initially thought that they would need this. It turned out that the time required to write such rules was deemed excessive, given that their function was largely documentary. Of course, another application such as a more conventional expert system might be likely to generate large numbers of such rules, necessitating machine-assisted debugging, or an application might have critical formal requirements that necessitated precise logical expression and checking.

- Of the various purposes listed above, we found the knowledge base was most helpful to the documentors, although this was probably due to the late stage at which CODE was introduced. The documentation team became very enthusiastic about CODE, because it greatly facilitated their task. Facilities were added to CODE to permit exporting cds and diagrams into publishing software, where it needed very little enhancement.

- Clearly, it would be desirable if CODE itself could take on as many as possible of the functions of the knowledge engineer. The present system takes no 'initiative', except to report certain discrepancies involving property inheritance. As we have seen above, often an important function of the knowledge engineer is to make suggestions, ask questions, or criticise knowledge as it comes in. Although some knowledge acquisition systems have tried to emphasise such a role so that they can be used without a knowledge engineer being present, the forms of knowledge that they can elicit must therefore be very restricted. Even so, we believe that a knowledge engineer should still review the result, which may need extensive reworking. We do not believe that the kind of knowledge captured in the Telos trial could be more than minimally criticised by any automatic system foreseeable within five years. However, some limited forms of semantic checking should be possible, somewhat like a good compiler; some of these will be incorporated in the next version of CODE. In any project where proper knowledge management is deemed essential, a knowledge engineer/documentor should be part of the team from the beginning.

- The use of CODE made it possible to refine the knowledge with a degree of clarity and economy of expression that would be extremely difficult using unstructured natural language supported by word processing. The Telos design
group was sufficiently impressed by the capability of the system and the importance of the associated ideas of conceptual analysis that it mandated that CODE should be used to capture all the design concepts for extensions to the Telos system.

Many of those involved in this trial now believe that the software development process ideally should include a knowledge management tool like CODE and a knowledge engineer as an integral component from the beginning. Initially, it would capture requirements; next it would record designer's evolving ideas, both individually and shared; and later it would record design and implementation decisions, and document the conceptual structure and terminology of the system. (It was, of course, only this later stage with which we have experimented). Designers should be taught to develop their ideas using the tool as an aid to their thinking, i.e. in conceptual analysis.

5 Future plans

A completely new version of CODE is currently operational [26, 27]. A potential focus is to experiment with the system for requirements analysis, a task in which we must typically deal with a large amount of new, vague, and often confusing concepts and terminology. We have also begun experimenting with CODE as a tool to assist in going from requirements, expressed as a knowledge base, to an object-oriented design, also expressed as a knowledge base, to an implementation, a third knowledge base [23]. Mappings are defined between knowledge bases to show how a concept in the requirements is implemented, if at all. Thus, we would integrate CODE directly into a software development environment so that an implementor of a design may access existing knowledge about the software component libraries they must use (and hence understand). Some of the knowledge could be automatically compiled into the implementation, e.g. potential object classes, variables and messages could be derived automatically from cds. Wang's thesis [28] illustrates how this could be done, using ObjectTime itself as the software development environment.

6 Conclusions

To the best of our knowledge, no existing tool provides as many of the knowledge management functions needed for software development as CODE, at least as an integrated easy-to-use package. Hypertext-like tools lack the AI component (e.g. inferencing and some natural language support), and most AI tools often lack good user interfaces e.g. a hypertext-like ability (rapid browsing) or display formatting. Many AI-oriented systems focus on 'strong' knowledge representations with certain inferencing abilities (e.g. 'term subsumption' languages or those based on logic) but lack many of the features of CODE that we deem essential. Tools designed for knowledge acquisition per se often are oriented to do only that, and often in some restricted domain, or focus solely on rule acquisition.

We have attempted in CODE to combine as many of those features that we feel are useful or essential for tasks like software development, based on years of practical experience with industrial software. We therefore believe that this trial of CODE in a real situation demonstrates some progress toward providing software developers with a knowledge management system that provides them with a collection of tools and techniques they can understand and adapt to their needs, without having to learn an excessive degree of formality or an excessively complex system. We have not read of a similar trial of such a system in an industrial setting.

A valid question would be 'how would CODE scale up?', e.g. to handle thousands of concepts. It is very hard to say, as we are unaware of any reported experience with the actual use of very large knowledge bases of this kind, let alone in an industrial setting. Only two very large knowledge base systems have been developed in research settings: the Cyc system [29] and the Botany system [30]. The former is by far the largest and most ambitious, with about 2,000,000 facts on general commonsense knowledge. The latter has tens of thousands of facts about 'first year' botany. Neither has been subjected to the same kind of trial as CODE, with real users reporting on their experiences, to the best of our knowledge. We would anticipate that, apart from speed limitations, the most serious problem would be in navigating and viewing; finding what you want amongst ten thousand concepts or properties, or trying to view parts of the knowledge base graphically on a small screen. Unfortunately, we are unaware of any published descriptions of such practical but critical problems with either of these systems and do not yet have our own experience. The field is simply too new.

Such systems can and should replace word processors (the current main choice) as the tool of choice for managing knowledge about systems. The Telos design group was sufficiently impressed by the capability of the CODE system and the importance of the associated ideas of conceptual analysis that it mandated that CODE should be used to capture all the design concepts for extensions to the Telos system. Only such realistic experience in actual industrial settings can serve as proof of the efficacy of new techniques.

7 Acknowledgments

This research has been directly supported by Bell-Northern Research, Cognos, Inc., and the URIF program of the Ontario government. It is derived from earlier research supported by Cognos, The Natural Sciences and Engineering Research Council of Canada, and the National Research Council of Canada.

The author would like to thank Dr. Dick Peacocke of the Computing Research Laboratory of BNR; Mr. Jim McGee, Director of the Advanced Services Group at BNR; the system engineers who patiently developed the knowledge base; Tim Lethbridge who performed much of the knowledge acquisition and added many features to CODE; and Ingrid Meyer and Ira Monarch who have been very helpful.

8 References


Software Engineering Journal  September 1995


© IEEE: 1995

The paper was received 20 February 1995.

The author is with the Department of Computer Science, University of Ottawa, Ottawa, Canada K1N 6N5.