A Development Methodology for *Composer*

Computer Support Tool for Academic Writing in a Second Language

Simon Shurville, Anthony Hartley and Lyn Pemberton
Brighton

This paper discusses issues in design methodologies for user centered writing environments, specifically the lessons we have learned in the early stages of developing *Composer*, a Cooperative Software system aimed at helping non-native speakers to write scientific texts in English. We believe that the development of an environment such as *Composer* depends crucially on paying attention to the dynamics of interaction within the development team. Central to the success of this interaction is a shared, evolving understanding of the team's objectives and the decisions made to meet them. Of the various representations used to foster this understanding, computer-based prototypes, however raw and unfinished, proved critical in enabling the whole team to engage with the dynamic and interactive nature of the system. This throws an interesting light on the current enthusiasm for paper prototyping.

1 Introduction

Professional scientists routinely use computer tools to write the reports and articles demanded by their discipline. Until recently they have done so without the aid of specialised tools, simply using general-purpose word processors. Recently researchers have started to produce software to support writers (Jansen 1994; Sharples/Goodlett/Pemberton 1992; Smith/Lansman 1992). Some studies have related development methodologies for writing environments to current developments in Software Engineering, such as rapid prototyping and user centered design techniques (Sharples/Pemberton 1990). However, accounts of the practicalities of managing the social and academic aspects of a necessarily interdisciplinary project are rare.

This paper discusses the lessons we have learned in the early stages of developing *Composer*, a writing environment aimed at helping non-native speakers of English to write scientific English within specific genres such as experimental reports, research articles and project proposals.

2 Designing a design methodology for *Composer*

The choice of a design methodology for *Composer* was inspired by recommendations drawn from Rittel and Webber's (1973) contributions to design research and Gerhard Fischer's (1992) contributions to the development of Cooperative Soft-

ware. Although these recommendations could not solve the problem of deciding upon a design methodology, they did offer a point from which to start understanding what the constituent problems might be. We describe them here in the belief that they can provide a suitable starting point for other developers of writing environments.

Rittel and Webber (1973) defined problems whose solutions call for elusive *political judgement* as being *wicked*. For designers, the major implications of recognising that a problem is wicked include the fact that they can neither pre-plan their design methodology nor assume that there is a definitive way to frame their problem. Instead, Rittel and Webber recommend, their design methodology and their frame of reference must co-evolve with their growing understanding of the problem and its context.

Gerhard Fischer (1992) treats the concept of wickedness in the context of developing what Rettig (1993) describes as Cooperative Software, computer programs that are driven by a person and which are intended to support that person's work, rather than autonomous artificially intelligent systems that would do that work instead of a person. In the main, Cooperative Software is developed to palliate the symptoms of the wicked problems found in activities where people follow professional or craft conventions. Fischer argues that an understanding of these conventions and of the practice of software development are pre-requisites for developing Cooperative Software. But the combination of these practices is likely to be too much for an individual to understand. So Fischer recommends that a team of developers needs to co-evolve a shared understanding of the relationship between the activity being supported and the software being developed to support it.

We will examine the features that define wicked problems and Cooperative Software and explain how the knowledge that we are aiming to ease a wicked problem with Cooperative Software has influenced our design methodology.

2.1 The nature of wicked problems

We will use the example of writing an introduction to a scientific report to introduce Rittel and Webber's concept of wickedness. Later, we will explain why designing a program to help writers to work in a second language is a wicked problem.

Consider the process of writing and evaluating an introduction to a scientific report. Theorists offer familiar guidelines to help an author to tame this problem: Swales (1990), for example, offers the *Create A Research Space* (CARS) strategy. Yet for all this theory, writing an introduction is a process which can never be fully tamed. Harmon and Gross (1996, 63), for example, note that the introduction "possesses the most formulaic structure" of any part of a report but cite Michaelson's observation (1986) that it remains "the most difficult and troublesome section to prepare". This is because writing an introduction involves making a series of decisions which each demand *elusive political judgement* on the part of an author. Let

us examine the *CARS* strategy in detail to appreciate why this wickedness is intrinsic to the problem – this will help explain why the most help an author can reasonably expect from a theorist is a set of guidelines rather than an algorithm.

The *CARS* strategy consists of three rhetorical moves. Move One: establish a research territory by showing that the general research area is important, interesting or problematic in some way (optional). Then introduce and review previous research in the area (obligatory). Move Two: establish a niche within that territory by introducing a gap in previous research, raising a question about it, or extending previous knowledge (obligatory). Move Three: occupy the niche by outlining the purpose or state the nature of the present research (obligatory), announcing the principal options (optional) and indicating the structure of the paper (optional).

Swales' strategy presents the author with an ordered list of features which should be present in her completed introduction. However, there are a number of reasons why it cannot be converted into an algorithm for designing, writing and evaluating the content of an introduction. The *CARS* strategy cannot tell the author how to: delineate an area of research; select items of literature to cite in order to establish the existence of a niche to be occupied; make that niche appear interesting to a prospective audience; select literature that will cement relationships with allies within that audience; decide which results to include or discard; evaluate whether the research actually occupies the niche it was designed to fill; or interleave the writing of the introduction with their experimental activities and with the rest of the paper. Comparing the process of writing an introduction according to the *CARS* strategy with Rittel and Webber's checklist of ten "distinguishing properties" of wicked problems should confirm that this is a wicked problem: (i) the problem has no definitive description; (ii) the problem solver cannot tell when she has completely solved the problem; (iii) the problem solver must decide whether a solution is good or bad rather than right or wrong; (iv) the perceived quality of a solution might increase or decrease over time; (v) a finite set of candidate solutions cannot be enumerated at the outset; (vi) each attempt to solve the problem incurs unrecoverable costs; (vii) the problem cannot be reduced to a familiar problem that has already been solved; (viii) the problem to be solved can be symptomatic of other problems and these may not come to light until a solution is implemented; (ix) the symptoms which identify the problem could be accounted for in many different ways; and (x) a solution has the potential to cause harm.

While Rittel and Webber introduced the concept of wickedness in the subject area of social planning, design methodologists now argue that wickedness is a concept that can be applied to the design of any artifact or system that interacts with human values (Cross 1984). Later, we shall explain how this concept applies to the problem of designing a piece of Cooperative Software to ease students' difficulties with learning to write in a second language. To prepare for that move we shall first rehearse Rettig's definition of Cooperative Software and discuss the recommendations which Gerhard Fischer offers to developers of Cooperative Software.

2.2 The wickedness of developing Cooperative Software

Rettig (1993, 24) gives the name Cooperative Software to computer programs where "the software and the person using it are partners in the task-at-hand, bringing complementary strengths and weaknesses to the job", citing Fischer's argument that humans bring to any job their common sense, a definition of a work goal and a plan for subdividing the problem. People also make frequent mistakes when working with formal systems such as mathematics, and sometimes lack relevant factual knowledge (as opposed to contextual understanding). Software, on the other hand, can make large amounts of knowledge available, can check the way people have reasoned within a formal system, and can help people to visualise information. What software cannot do is bring its own sense of purpose or context to a job. Other researchers, notably Silverman (1992), have worked from similar premises, applying ideas from areas such as decision making theory to software systems. Hence the 1990s have witnessed the development of a variety of cooperative systems that reject the idea of using *strong artificial intelligence* to perform a task for a person. Instead they favour adapting a hybrid of *lightweight artificial intelligence*, hyper-media and visualisation technologies to allow a person to enter a more balanced relationship with her software.

These Cooperative Systems are particularly suited to helping people work with wicked problems. We have seen that wicked problems are solved by building an understanding of a problem in its context. This is a process that requires access to pertinent information and demands contextual and social knowledge. A fruitful partnership between a person and a Cooperative Software system relies on the fact that the person can direct the way they come to understand the problem by browsing their software for pertinent information. The Cooperative Software will often make available mainstream technologies, such as a spreadsheet or word processor, so that the person may solve parts of her overall problem in the context of a system that is targeted at a specific type of job. Meanwhile, the Cooperative Software can attempt to use its lightweight artificial intelligence to critique potential errors in the person's judgement or to direct them away from over-used problem solving strategies (Silverman 1992).

Wicked problems are often found within activities such as engineering, town planning or document drafting, where conventions have evolved in a particular social context. A system must offer functionality and interface features which are in harmony with current working practices.

Yet developing Cooperative Software to take account of the conventions and practices of a type of job is a wicked problem in its own right. This can be illustrated by exploring the *niche* that *Composer* will occupy within the field of Computer Assisted Language Learning. First, it is unlikely that a single developer will embody sufficient expertise across each of the pedagogic and technological dimensions of the niche that are necessary to produce a sound specification (Fischer 1992). Second, since the domain is not only interdisciplinary but also novel, the

end-users have not have previously worked with similar software. It is therefore unlikely that an expert in either pedagogy or computing will be able to envisage the true needs of these users. It is still less likely that an expert in either field can predict any new problems that delivering a piece of collaborative software might bring to light, including the pedagogic harm that over-reliance on an untried feature might cause.

For these reasons it is important that the design of a system like *Composer* is regarded as a group exercise, involving various domain experts who are familiar with an aspect of the pedagogic or technical dimensions of the software. It is also important that the initial understanding of the problem and its practices is evaluated by a sample of end-users of the Cooperative Software. This procedure should be repeated in an iterative fashion throughout the life of the project. Initially, as we shall see, these end-users can evaluate statements of intent or sketches of a working system. These can lead to more formal prototypes and eventually to working code. This is especially problematic because the results of evaluating each stage of understanding a problem, or its encoded artefacts, must be disseminated to a team who think in terms of different terminologies, goals and practice. It also raises the question of where these end-users will be found.

Rittel and Webber's (1973) checklist of necessary and sufficient conditions for wickedness reminds us that an iterative methodology of specification, prototyping, evaluation and consequent re-specification is unlikely to proceed in a deterministic fashion. It might, for example, turn out to be the case that improved library resources could help students more than a piece of Cooperative Software. This is especially true for a project with temporal and budgetary constraints. One reason for this is that the initial choice of members of a design team together with the disciplines which they represent will affect the way in which the problem domain is perceived. Hence the specification may ignore a valuable dimension of the problem because the team lacks some appropriate domain expertise. A second reason is that evaluation of the way end-users interact with the software and of what they learn from its use must take place in a finite amount of time. Yet it is possible that problems will be missed because they cannot come to light during the period that has been allocated. Hence a score that the evaluators award to a certain feature of the software might increase or decrease if the evaluation were given more time. A third reason is that a badly specified prototype might harm the academic progress of its end-users in some way. Hence, to paraphrase Rittel and Webber (1973), *the design team has no right to be wrong*. In sum, designing a piece of Cooperative Software in an institutional setting can be complicated by issues which include: staffing policies that are set externally to the design team, the availability of domain experts and end-users and the moral responsibility which members of an academic institution owe to their students. These are problems which have analogies in the development of Cooperative Software, which is a wicked and complex problem because each attempt to find a solution incurs costs and may lead to unexpected consequences.

3 The development methodology for *Composer*

In the previous section we described the need for an interdisciplinary development team for *Composer* and highlighted the importance of evolving a shared understanding of the problem at hand. In this section we will describe the areas of expertise that are represented within the team, explain how we set out to build this shared understanding and summarise the prototyping activities that we have performed.

3.1 Orchestrating a development team and fostering communication

We assembled an interdisciplinary development team for *Composer* which includes eight people. These people can variously be described as *teaching and writing people*, *computers and writing people*, *computer people* or *graphics people*. The teaching and writing people bring expertise in second language acquisition, Teaching English as a Foreign Language (TEFL), designing and teaching courses in English for academic purposes, translation and technical writing. The computers and writing people bring expertise in artificial intelligence in education, student modeling for second language acquisition, language engineering, and developing writing environments. The computer people bring expertise in object-oriented rapid prototyping and human computer interface design. Finally, the graphics people bring expertise in information design, especially in evaluating the legibility of text.

These team members are organised into a daisy-chain formation of sub-teams to address the following circle of tasks: specification, interface design, prototyping, evaluation and re-specification. Each of these sub-teams contains one of our TEFL teachers, who acts as a *gate keeper* providing access to students of English as a foreign language. The membership of each team overlaps with the teams performing the tasks preceding and following its own. For example, the specification team overlaps with the interface design team which in turn overlaps with the prototyping team.

In order to communicate effectively we needed to learn about each other's disciplines and interests. In other words we needed to be rapidly inducted into each other's discourse communities. With this in mind we wrote critical literature surveys of our fields to distribute amongst ourselves. We also distributed what each of us believes to be key papers within our fields. The teaching and writing people's survey helped us understand why we should avoid text-analysis programs with canned response and instead develop a program to "train the editing process" (Pennington 1992), enabling students to operate autonomously. The computers and writing people's survey directed our attention towards pedagogic texts, especially toward Swales and Feak (1994), and towards the Cooperative Software paradigm. The computer people highlighted the utility of object oriented rapid prototyping in the context of solving a wicked problem (Connell/Shafer 1995). The graphics people helped us to understand how the principles of information design could be used

to help us design a comprehensible interface that makes good use of limited screen space (Tufte 1990). For consistency each member of the team was issued with a folder containing the same key papers and literature reviews. This was felt to be important because the development team are distributed across different University sites and thus do not have shared access to a common library building.

The final move in our orientation involved making use of an existing program called the *Writer's Assistant*. This is a writing environment which one of us helped to develop at the University of Sussex (Sharples et al. 1992). The *Writer's Assistant* contained many features that could be adapted to help students to learn to write in a foreign language, including a rich set of external representations of the interim processes of writing (Pemberton/Sharples/Shurville 1996). The *Writer's Assistant* provided a useful and concrete focus for the initial discussions between the computer people and the teaching and writing people.

3.2 Learning from students and teachers of English in an institution for higher education

Once we felt we had learned enough about each other's interests to start to understand our problem it was clear that we needed to learn about the perceived needs of our end-users. We identified two distinct sets of people who could be described as end-users of Composer, non-native students of English who are learning to write documents in higher education and teachers of such students.

Studies using questionnaires and focus group meetings with teachers and potential users were carried out to analyse the needs of second language learners in the classroom, with the aim of discovering which problems of second language writing might be amenable to computer support. Our informants allowed us to distill a number of requirements for both the functionality and the interface style of such a support system.

The overwhelming desire amongst the students was for a system compatible with the word processors they already used, which would support them in carrying out their own writing tasks in their own way. There was no enthusiasm for a system which imposed its own agenda in the form of artificial writing tasks carried out via an externally prescribed process. The students' aim in writing in English was to emulate the sorts of texts written by native speakers and there was a demand for models of such texts to be made available on-line in a variety of forms and for a variety of purposes. Our target users already have a relatively high degree of competence in English grammar and grammar checking was not requested. There was enthusiasm, though, for facilities to check for usage and constructions that were particularly troublesome for the individual or for particular language groups. These facilities should be open to modification by the writer. All the students worked in different ways, using different tools and did not want to be faced with changing their styles of working. Since they would be concentrating on their main task, that of composing, they wanted the use of any teaching aids and tools to be optional.

However, when they chose to use the tools they did not simply want programs to point out their errors after they had made them. They preferred software which would allow them to learn and improve as writers, not simply facilities to correct aspects of a single piece of writing. The interaction model suggested by these requirements is close to the Cooperative Software model.

The teachers we surveyed work within a variety of subject areas, e. g., computer science and business administration and expect their students to learn how to write in a variety of genres, e. g. scientific and business reports. They also teach students with different first languages. The teachers suggested that *Composer* should be adaptable so that it can be used in different combinations of genre, subject area, and eventually in different combinations of L1 and L2. The teaching and writing people confirmed that it would be prudent to include modules within *Composer* that would allow teachers to adapt its knowledge base to take account of how subject areas and genres vary across academic departments, institutions and languages. It was also agreed that a scientific report would serve as a good *proof of concept*, especially since there is a wealth of academic material devoted to this genre. The choice of the introduction as our initial area for explanation seemed a natural one given the existence of guidelines such as *CARS* that appear suited to being adapted to a tutorial context.

After evaluating our first round of focus groups and questionnaires it was confirmed that we would avoid overly prescriptive styles of interaction between the student and the computer and algorithms that perform tasks such as text-ordering for the student. Instead we would concentrate on developing a *generic* piece of Cooperative Software that could be linked to standard word processors on a variety of platforms via a technology such as *object linking and embedding*. The task assigned to this addition to a standard word processor was to present structural information, models and linguistic points that could be reflected upon and learned from in the context of a *live* document that a student had been asked to write by a tutor.

We began to understand that developing *Composer* is less a problem of finding intelligent algorithms, which were at the core of the *Writer's Assistant*, than of finding pertinent information and presenting it to students in an intelligible way. The task we faced next was to sketch *Composer's* functionality and appearance in a suitable format for an initial evaluation by our own team members and a subsequent round of end-user evaluation.

3.3 Prototyping with white boards, paper, HyperCard and VisualWorks

It is becoming accepted that "prototyping provides a communication basis for discussions among all the groups involved in the development process" (Budde/ Kautz/Kuhlenkamp/Züllighoven 1992, 90). However, the *fidelity* with which a given prototype should resemble a final product during the development process remains a burning issue within the discourse community of Human Computer

Interaction (Rudd/Stern/Isensee 1996). In this section we describe our rationale for prototyping at various levels of fidelity during the initial stages of the project. We then present some illustrations of our prototypes and discuss what we learned about choosing an appropriate level of fidelity.

It is possible to prototype the interface and functionality of a system at many levels of realism, from simple hand-drawn sketches to prototypes which look like the real thing. The choice of an appropriate prototyping medium at each point in a project involves estimating how much imagination is required from the designers to propose a new design concept and how much imagination is required on the part of the evaluators in order to provide useful feedback on that design concept.

We began our prototyping by using a white board, moving to paper-based prototypes before implementing low and high fidelity prototypes in *Hypercard* and *VisualWorks*. We needed to begin with pictures of a system that could be discarded at minimal cost as the team proposed design concepts and evaluated and synthesised them on the basis of their own experience and expertise. We also reasoned that the team members would be able to understand and evaluate even very low fidelity prototypes on white board and paper.

Figure 1 shows a paper prototype of one of the views of a writer's text which *Composer* will present. The cascading menus within the window labelled *Scientific report: my report* indicate the context sensitive help that will be made available in each section of the scientific report that a student is composing.

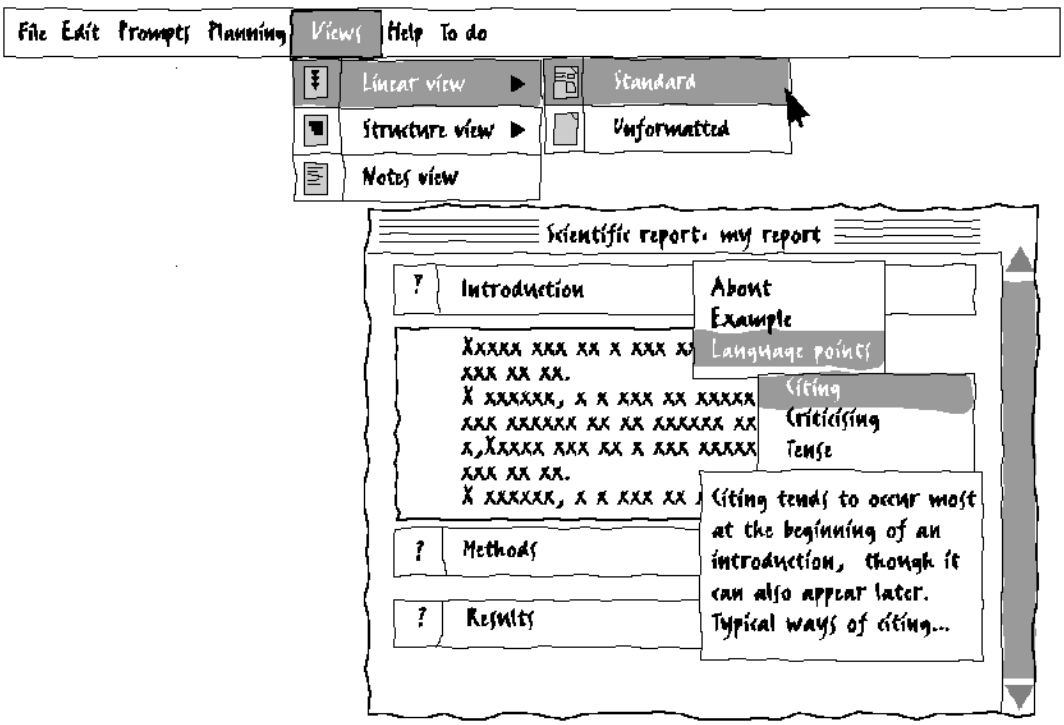

Fig. 1: A 'paper prototype' of one of the views presented in *Composer*

The fine grained details and rationale behind these prototypes is beyond the scope of this paper (please see Pemberton/Shurville/Hartley 1996 for more deatils). What is important is that the computer people and the computers and writing people in the development team intuitively understood what these prototypes represented: they were familiar with the conventions of the software domain. They failed to notice that the teaching and writing people in the team did not understand what the prototypes represented. The computer people had failed to realise that a computer implementation, however simple, may be needed by specialists from another field very early on in the development process. This realisation dawned when one of the computers and writing people wrote a low fidelity version of the same part of the interface in *Hypercard*. This is illustrated in Figure 2, which shows a structure to guide the writing of an introduction according to the *CARS* methodology.

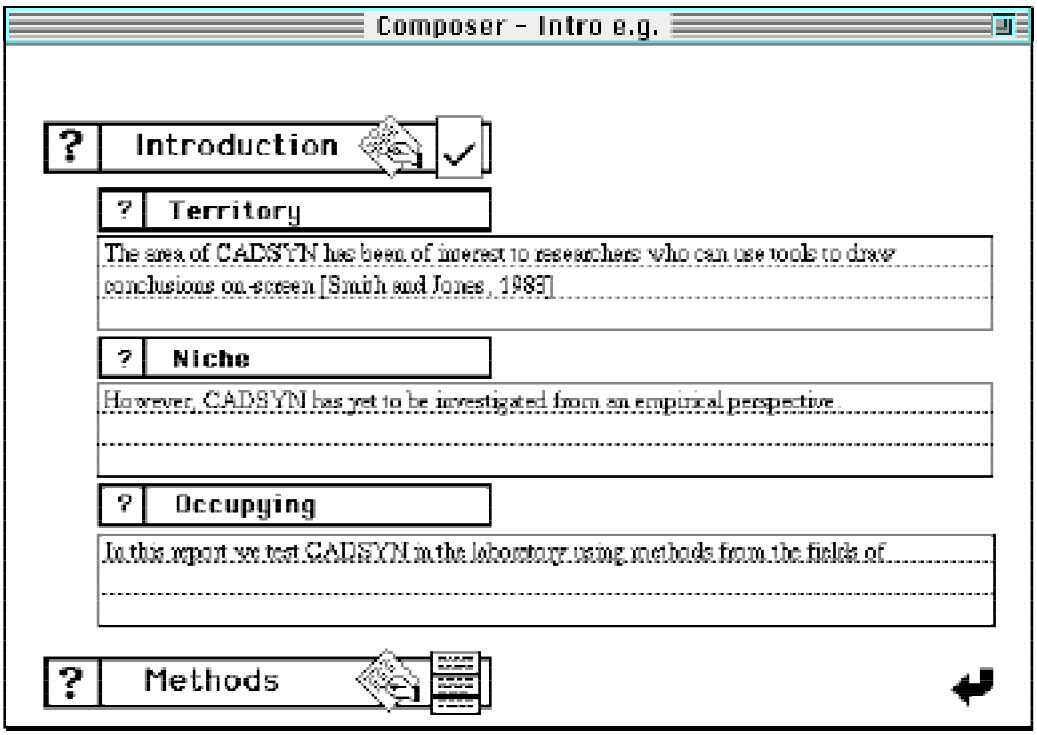

Fig. 2: A Hypercard prototyp for part of Composer showing support for the writing of an introduction according to the CARS methodology

The next lesson we learned from evaluating the *Hypercard* prototype was that some of the buttons and menus were badly positioned. We explored alternative arrangements in a higher fidelity prototype in *VisualWorks*, a visual programming environment with *Smalltalk* as the *scripting language.* This environment allowed us to prepare a more realistic version of *Composer* to present to our students and teachers and also provided an opportunity to begin to model the class libraries we would need to implement *Composer.* In other words, we made the transition form *throwaway* to incremental prototyping. This prototype is illustrated in Figure 3.

This illustrates the next step in writing an introduction where the student wants to see the results of following each of the three steps of the *CARS* methodology in one space so that she can start to edit them into a series of linked paragraphs.We re-arranged the various buttons and menus that provide help and information into the format of standard *palettes* that appear at the beginning of each section of a document.

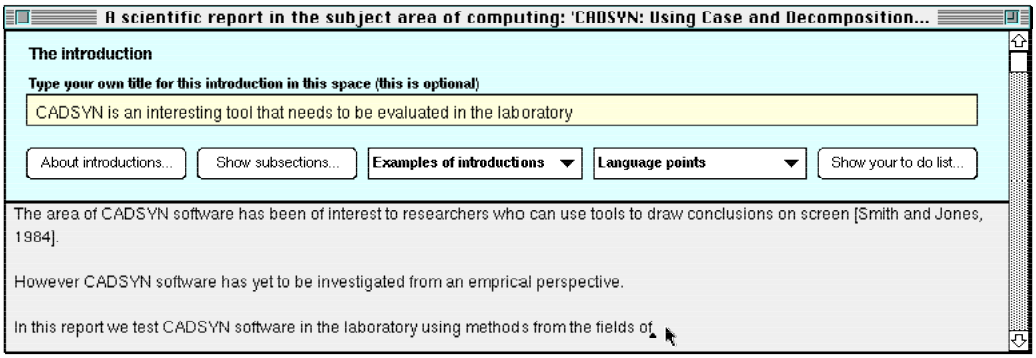

Fig. 3: A VisualWorks prototype for part of Composer showing support for the writing of an introduction according to the CARS methodology

4 Summary

We believe that the development of an environment such as *Composer* depends not only on technical issues but also, crucially, on paying attention to the dynamics of interaction within the development team and between the developers and end-users. Central to the success of this interaction is a shared, evolving understanding of the team's objectives and the decisions made to meet them. The representations used to foster this understanding may take various forms. We have learned that the use of computer-based prototypes, however raw and unfinished, is vital if the whole team is to be enabled to engage with the dynamic and interactive nature of the system. This throws an interesting light on the current enthusiasm for paper prototyping.

References

Budde, Reinhard/ Kautz, Karlheinz/ Kuhlenkamp, Karin/ Züllighoven, Heinz (1992): What is Prototyping? In: Information Technology and People 2/3 (6), 89-95

Connell, John/ Shafer, Linda (1995): Object Oriented Rapid Prototyping. Englewood Cliffs: Yourdon Press

Cross, Nigel (ed.) (1984): Developments in Design Methodology. Chichester: John Wiley

Fischer, Gerhard (1992): Domain-Oriented Design Environments. In: Proceedings of the Seventh Knowledge-Based Software Engineering Conference. Washington: IEEE Computing Society, 204-213

Harmon, Joseph/ Gross Alan (1996): The Scientific Style Manual: A Reliable Guide to Practice? In: Technical Communication 1 (43), 61-69

Jansen, Carel (1994): Computerised Writing Aids: Do They Really Help? In Steehouder, Michaël/ Jansen, Carel/ van der Poort, Pieter/ Verheijen, Ron (eds): Quality of Technical Documentation. Amsterdam: Rodopi, 239-248

Michaelson, Herbert (1986): How to Write and Publish Engineering Papers and Reports. Philadelphia: ISI Press

Pemberton, Lyn/ Shurville, Simon/ Hartley, Anthony (1996): Motivating the Design for a Computer Assisted Environment for Writers in a Second Language. In: Diaz, Arantza/ Fernandez, Isabel (eds.): Computer Aided Learning and Instruction in Science and Engineering, Berlin: Springer-Verlag, 141-148

Pemberton, Lyn/ Sharples, Mike/ Shurville, Simon (1996): External Representations in the Writing Process and How to Support Them. In: Self, John (eds.): Euro Artificial Intelligence in Education 96, Forthcoming

Pennington, Martha (1992): Beyond Off-the-Shelf Computer Remedies for Student Writers: Alternatives for Canned Feedback. In: System 4 (20), 423-437

Rettig, Marc (1993): Cooperative Software. In: Communications of the ACM 4 (36), 23-28

Rittel, Horst/ Webber, Michael (1973): Dilemmas in a General Theory of Planning. In: Policy Sciences 4, 155-169

Rudd, Jim/ Stern, Ken/ Isensee, Scott (1996): Low vs. High Fidelity Prototyping Debate. In: Interactions, January 1996, 77-85

Sharples, Mike/ Goodlet, James/ Pemberton, Lyn (1992): Developing a *Writer's Assistant*. In: Hartley, James (ed.): Technology and Writing: Readings in the Psychology of Written Communication. London: Kingsley, 209-220

Sharples, Mike/ Pemberton, Lyn (1990): Starting from the Writer: guide lines for the design of user-centred document processors. Computer Assisted Language Learning 2, 37-57

Silverman, Barry (1992): Critiquing Human Error: A Knowledge-Based Human-Computer Collaboration Approach. London: Academic Press

Smith, John/ Lansman, Marcy (1992): Designing Theory-Based Systems: A Case Study. In: Bauersfield, Penny/ Bennett, John/ Lynch, Gene (1992): CHI'92 Conference Proceedings. Reading MA: Addison-Wesley, 479-488

Swales, John (1990): Genre Analysis: English in Academic and Research Settings. Cambridge: Cambridge University Press

Swales, John/ Feak, Christine (1994): Academic Writing for Graduate Students: A Course for Non-native Speakers of English. Ann Arbor: University of Michigan Press

Tarone, Elaine/ Yule, George (1989): Focus on the Language Learner. Oxford: Oxford University Press

Tufte, Edward (1990): Envisioning Information. Cheshire CN: Graphics Press