AN ARCHITECTURE FOR A COLLABORATIVE AUTHORING SYSTEM

Collaborwriter

KEITH McALPINE BSc (J. Hons)

Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

June 1996

THE UNIVERSITY OF ASTON IN BIRMINGHAM

AN ARCHITECTURE FOR A COLLABORATIVE AUTHORING SYSTEM

Collaborwriter

KEITH McALPINE BSc (J. Hons)

Doctor of Philosophy

June 1996

Synopsis

Collaborative working with the aid of computers is increasing rapidly due to the widespread use of computer networks, geographic mobility of people, and small powerful personal computers. For the past ten years research has been conducted into this use of computing technology from a wide variety of perspectives and for a wide range of uses.

This thesis adds to that previous work by examining the area of collaborative writing amongst groups of people. The research brings together a number of disciplines, namely sociology for examining group dynamics, psychology for understanding individual writing and learning processes, and computer science for database, networking, and programming theory. The project initially looks at groups and how they form, communicate, and work together, progressing on to look at writing and the cognitive processes it entails for both composition and retrieval.

The thesis then details a set of issues which need to be addressed in a collaborative writing system. These issues are then followed by developing a model for collaborative writing, detailing an iterative process of coordination, writing and annotation, consolidation, and negotiation, based on a structured but extensible document model. Implementation issues for a collaborative application are then described, along with various methods of overcoming them. Finally the design and implementation of a collaborative writing system, named *Collaborwriter*, is described in detail, which concludes with some preliminary results from initial user trials and testing.

Keywords

CSCW, collaboration, groupware, negotiation, writing

Acknowledgements

This thesis would not have been completed without the help, advice and encouragement of a number of people.

First, I would like to thank my supervisor Paul Golder for all his help and suggestions made throughout my PhD. Secondly I would like to thank Keith Lander for giving me my first experiences of the wonders of writing word processing systems and who thought I would never want to see another text editor afterwards! Thirdly I thank Mark Roseman and the team at the University of Calgary for all their work on the GroupKit environment, without which I would be swarming in a mist of network protocols.

My sincerest thanks also go to my family, who have always supported and encouraged me throughout my PhD, and without whom it would never have been completed.

Finally, my greatest thanks go to my girlfriend who listened through many a day of me talking about social protocols, group behaviour and timing problem issues, who proof–read this text and who had a fairly stressed year as I battled to finish the thesis while holding down a full–time job.

List of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer supported cooperative work (CSCW) is an area of computing which has grown rapidly during the past ten years. Improved communications bandwidth, more powerful personal computers and the movement from national to global markets are each elements which have encouraged a change of emphasis from individual–based, localised solutions to group–based, global solutions in the computing field.

Business, for example, has become increasingly global. Other than traditional import and export transactions, Guy and Mattock (1991) note that shifts to low–cost manufacturing sites, take–overs and mergers have rapidly increased in number. "*Transnational companies*" (Bartlett & Ghoshal, 1992) have subsequently emerged and with them a new management practice treating world–wide operations as an integrated and interdependent whole.

Initially, basic strategies for such companies were decided at face–to–face meetings of senior management. Now, in the search for an improved competitive edge, such decision–making tasks need be taken at a more regular operational level. With decentralised industries, a person may now find that they are working in project teams encompassing people not just from a specific plant, but from multiple establishments situated around the world. These people must coordinate their efforts to ensure the efficiency and effectiveness of their joint resources in an increasingly competitive marketplace.

On a cultural level, people are now finding that they are able to have more flexible working practices in both a time and geographic nature. Many companies offer a system of "*flexitime*" with some extending this to enable people to work from their home. Powerful laptop computers and communication links also enable people to work "*in the field*" or from alternate sites.

This increase in flexibility and collaborative working introduces a number of problems, such as:

- **An increase in coordination overhead.**

  Groups of people, such as a group with members spread across America, Europe and Japan, could be working 'together' on a project in both separate time zones and without ever meeting one another. In order to do this effectively, their work needs to be carefully coordinated to ensure reduced work overlap and increased awareness of what each person is doing.

- **Cultural difficulties in international collaborative efforts.**

  In a global business environment, individuals require new skills and competencies in order to work and deal with people from other countries and with different cultural priorities (Mead, 1994). Thus in addition to linguistic difficulties, different cultures place different emphases on decision–making, negotiation strategies and general group working.

- **An increase in software development complexity.**

  Most of the focus of software development in recent years has been based around applications written for *personal* computers. That is, applications designed and written to help and improve an individual's productivity. Collaborative applications on the other hand, greatly increase the complexity of an already difficult development task. New tools, models, frameworks and methodologies are needed to aid the programmer in developing collaborative applications. Human factors studies need also to be performed on examining how tools can be improved to work within *groups* in the same way that studies have been performed on how *individuals* work with a tool. These studies are further complicated when regard is made of the composition and size of a group and the task it is asked to perform – whilst there are many 'types' of individual, there are a great many more types of group.

1.1 Rationale and Scope of the Dissertation

This thesis focuses on one area of group working, namely collaborative writing. Most computer–supported writing systems are based around the single user and, where a tool provides facilities for multi–user use, this is limited to features such as basic annotation or version control. Indeed in professional publishing systems, such as in the Interleaf™ publishing package, multi–user facilities consist only of annotation and version control and a check–in/check–out method to control writing cross–over conflicts. These systems do not help writers during other stages of the writing process, such as the initial coordination stage or the consolidation of completed work.

As there are many types of group, so there are many forms of writing. Different authoring scenarios range from taking notes in a group meeting, to changing code in a multi–user programming environment, to the writing of large fluid documents by many authors in a large organisation. The writing practices adopted by authors can also vary, ranging from tight synchronous collaboration where the authors work closely on a piece of text in real–time to loose asynchronous collaboration where authors only merge their work at specific points in a project.

Collaborative writing tools can be developed to focus on specific writing tasks or attempt to be versatile enough so as to be used in a number of different ways. This dissertation examines the area of collaborative writing through merging a number of disciplines, namely sociology for examining group dynamics, psychology for understanding individual writing, learning processes and human–computer interaction, graphic design for typography and computer science for database, networking, text–processing and programming theory.

1.2 Objectives of the Dissertation

The main objective of this thesis has been the development of a tool for collaborative writing. Much of the research on collaborative computing regards the users as working cooperatively for small–scale tasks. One of the assumptions of this thesis, however, is that individuals will work together although not necessarily in cooperation, with conflict and negotiation playing a

major role in the process. Indeed Vámos (1991) notes that in complex group processes all forms of cooperation are based on negotiation of one type or another.

This thesis is concerned therefore with the development of a suite of collaborative tools to aid authors in all stages of the writing process – from initial conception through to finalised work – and which together form a complete writing environment. To accomplish this objective a number of stages and subordinate objectives have been identified:

1. an investigation of group processes with specific regard to decision–making, cooperation and conflict;

2. an investigation of the writing process from cognitive, structural and typographic viewpoints;

3. the development of a writing model for both synchronous and asynchronous group writing;

4. the development of a software architecture for collaborative applications;

5. the development of user–interface widgets to improve group awareness;

6. the development of applications based on the group writing model; and

7. user trials of the developed system to test its assumptions and implementation of these subordinate objectives.

1.3 Structure of the Dissertation

The thesis is divided into seven chapters, with this chapter providing an introduction and overview to the rest of the dissertation.

Chapter two forms the start of the literature review and examines the area of cooperative working from primarily sociological and psychological perspectives. This includes a general overview of the characteristics of a group and the processes it goes through, such as the group's formation, size, social dimensions, participant roles and communication structures.

The chapter then presents an overview of the decision–making process and details some of the problems and solutions faced by the decision–maker for both individual and group situations. Following from this is an examination of group cooperation and group conflict along with a set of assertions and assumptions about conflict. Finally, the chapter closes with a short review of computer support for groups.

Chapter three forms the body of the literature review on writing, looking at the subject from typographic, structural and cognitive viewpoints. The chapter opens with an examination of text structure and typography, followed by research on textual mark up. The art of writing is then presented, from the design of text to improve reader access and retention, to cognitive models of writing, to computer systems which can aid authors in the writing process. Finally the chapter gives an account of collaborative writing, starting with collaborative writing practices through to models for collaborative writing and ending with a description of computer systems designed to support such writing activities.

Chapter four is concerned with the design of a collaborative writing tool. It begins by listing the main issues which need to be adopted by an authoring tool, with different emphases given to the issues depending on the writing task to be performed. This is then expanded on to present a model for collaborative writing, detailing an iterative process of coordination, writing and annotation, consolidation and negotiation which is based on a structured but extensible document model. The chapter finishes with a description of the main communication flows between authors during the different stages of the writing process, and how different forms of communication are required during the various stages of the writing process.

Chapter five describes the main implementation issues concerning any collaborative application. Beginning with a description of different architectures for computer–based collaboration, a groupware toolkit called GroupKit (Roseman & Greenberg, 1992) is described with details of the collaborative model and programming abstractions the system uses. A method for developing collaborative applications is then presented under the title of

environmentally–aware software. Finally, the chapter concludes with a description of the development of a small collaborative application.

Chapter six details the development of a collaborative authoring tool called Collaborwriter. The chapter first details the run–time architecture of the system, followed by describing extensions to the GroupKit toolkit for locking and transaction management and for new user–interface groupware widgets. To close the system is described in detail, with sections describing the main tools (the project management tool, the authoring tool, the negotiation tool and the consolidation tool) used in the environment.

Chapter seven concludes the dissertation with some results from initial user trials of the system, showing how the tools link in with the writing model developed in chapter four. Finally it describes those areas where the system could be expanded and where further research and work is required.

Chapter 2

Cooperative Working

2.1 Introduction

For any form of computer–supported collaborative tool, there is a need to look at the environment it is to be used in and the processes it may change. Changes in work practices may be for better or worse, so the work practices should be first examined before they are potentially changed. In this way, good practices might be enhanced by a tool and bad practices diminished.

With regard to the area of collaborative authoring, the main practices to be examined are the basic elements and processes involved in group behaviour and the process of writing itself. Through an understanding of both these topics may come the required insight into what needs to be adopted by a collaborative authoring tool.

This chapter will first look at group processes, primarily from a sociological and psychological perspective, focusing on aspects such as its formation and size, social dimensions, participant roles, and communication structures. The decision making process will then be examined, relating to both individuals and groups, followed with a discussion on cooperation and conflict. Finally, the latter part of this chapter will look at how computer systems have been developed to help support some of the group aspects and problems with decision making in general.

2.2 The Group

A group can be defined as *"a collection of two or more interacting individuals with a stable pattern of relationships between them who share common goals and who perceive themselves as being a group"* (Forsyth, 1983). Thus, for a group to be more than simply a 'collection of

individuals' the members need to interact with one another in some way (not necessarily in a face–to–face situation), share one or more common motives or goals, and actually feel that they are each part of a group.

2.2.1 Group Formation

Znaniecki (1939) and Sherif (1954) identified a number of features which occur in the formation of a group. First, the members each share one or more common goals for determining the direction in which the group moves. Next, a set of norms are developed to define boundaries within which interpersonal relations are established and activities carried out. Thirdly, the member roles are stabilised and the group becomes differentiated from others. Finally, a network of interpersonal attraction develops on a basis of likes and dislikes of members for one another.

These theories were further developed by Parsons (1961) into the AGIL scheme of groups, consisting of: *adaptation* (generate the skills and resources needed to reach the group goal); *goal attainment* (exercise enough control over the membership in order to reach a common goal); *integration* (set rules to coordinate the activity with enough feeling of solidarity so that the group stays together); and *latent pattern maintenance* (give the group members a common identity and commitment to group values that is unique to their group).

2.2.2 Group Development

From ideas first developed by Tuckman (1965), Tuckman and Jensen (1977) argue that a group is a fluid entity and goes through a variety of changes over time, and have identified five main stages in the development of a group. These stages are:

- **Forming**

  In the earliest stages of group development, members are most concerned with being accepted within the group and learning more about the group and its situation. The stage is marked by polite and inhibited behaviour and information gathering on the part of the

members, and is concluded once the individuals come to think of themselves as members of the group.

- **Storming**

  As the group matures and individual members become more secure, there is a period of conflict where members confront their various differences and personal goals are revealed. Members may resist the control of other group members and early relationships, established in the forming stage, may be disrupted. Group direction and conflict management are key to this stage as, unless the conflicts are resolved, the group may disband. This stage ends once conflicts are resolved and the group's leadership is accepted.

- **Norming**

  After the turbulent previous stage, the group develops some consensus regarding roles, status and procedures. The members start working together, developing close relationships, feelings of camaraderie, and shared responsibility. Hostility and conflict is lower and group cohesion increases, with the stage being complete when the group members come to accept a common set of expectations which constitute an acceptable way of doing things.

- **Performing**

  The group has fully developed and is now ready to work, questions on group relationships and leadership having previously been resolved, so that the group can devote its entire energy to the task at hand. Not all groups develop to this stage, however, but may become bogged down in an earlier and less productive stage.

- **Adjourning**

  Finally, a group may disband. This may be due to the group having completed its work, (as in a group created for a specific project), where the end is abrupt. Alternatively the group may disband gradually as it disintegrates, either through members leaving or because the norms which had been developed are no longer effective for the group.

A group can be in any of the stages of development at any given time, with the time in each stage highly variable. Stages may also be combined as deadline pressures force groups into action (Gersick, 1988) or, because certain issues have not been addressed at earlier stages, the group may not be working at full efficiency. For example, people may be pulling in different directions because the purpose of the group has not been clarified, nor its objectives agreed (Huczynski & Buchanan, 1991).

2.2.3 Group Characteristics

Not all groups are alike, with each group having its own composition and structure. This relates to aspects such as type of group, group size, group norms, individual roles, and the group's communication network.

2.2.3.1 Types of Group

Groups can generally be categorised as either formal or informal. *Formal groups* are created by the organisation and can be either a command group, based on the organisational power hierarchy, or a task group, which is designed and created around specific tasks, the membership is generally for a specific length of time, and individuals are chosen to be members due to their specific expertise. Through formal groups an individual can attain a formal role in a company. Individuals may also be members of many formal groups at once in an organisation (Likert, 1961), such as in a research group, a standing committee, and a product development team.

Formal groups take direction from an outside manager or group, except in the case of self–regulating work groups. These groups are relatively free to control their own work and have proven successful in non–routine and creative tasks. The groups also have greater member satisfaction, reduced people turnover, increased productivity, and reduced costs when compared to a managed group (Pearce & Ravlin, 1987).

Informal groups are built through the social interactions amongst people, not through the direction of management. An informal group can be either an interest group, formed out of

shared interests between members, or a friendship group, formed out of mutual binding between members. Informal groups can be powerful entities due to their coherence and psychological solidarity and, if it can be influenced, can be a great asset when the goals of the group and of management are in line. As an informal group grows, there tends to be an increase in hostility towards some outgroup, usually associated with an increase in affection for members of the in–group (Sherif, 1956). Informal groups are also more stable than formal groups if the members know one another for a length of time, with members having higher morale and increased productivity (Van Zelst, 1952). Once established, however, new membership to the group can be difficult as it develops its own practices and norms.

2.2.3.2 Group Size

The size of a group can have important ramifications on its productivity. Larger groups are more likely to include individuals with specific skills, causing specialisation of labour, yet leading to the individuals feeling more anonymous. Individuals subsequently become less satisfied and feel less accountable for their actions (Schellenberg, 1959). Group size also affects communication amongst the members. As the size increases, a smaller percentage of individuals participate in group discussions, due to members' heightened fear of participation and there being physically less chance or time for individuals to express themselves (Huff & Piantianida, 1968).

Bales and Borgatta (1955) found that even–numbered groups had a higher rate of showing antagonism and disagreement and a lower rate of asking for suggestions and showing agreement than odd–numbered groups. This was because the group could in turn split into sub–groups, or "camps", of equal size and remain deadlocked over an issue. Bales (1954) determined that the optimum small group size was 5 because:

- strict deadlock was not possible with an odd number of members;
- it formed a 3–2 split so that the minority was not one isolated individual; and
- the size was large enough for members to shift roles easily, with an individual able to withdraw from an awkward position without having the related issue resolved.

This was further expanded by Slater (1958), who concluded that a group size of five members was optimum for dealing with mental tasks in which the group members collect and exchange information, and then make a decision based on the evaluation of it. Osborn (1957) however, suggested that the optimum size of such groups was between five and ten members, as in a brainstorming activity.

2.2.3.3 Group Norms

During the development phase, groups tend to both form and conform to a set of group norms. These norms are informal rules that guide group members' behaviour, representing shared ways of viewing the world (Hackman, 1991). Norms can be developed in different ways. Feldman (1984) summarised four ways in which group norms could be developed:

1. Precedents set over time (e.g. the seating location of each group member around a table);
2. Carryovers from other situations (e.g. professional standards of conduct);
3. Explicit statements from others (e.g. told how to do a task in a certain way); and
4. Critical events in group history (e.g. a secrecy norm after a loss occurring due to someone divulging company secrets).

Group norms can have profound effects on an individual's judgement and perception. For instance, Sherif (1961) has noted that group members tend to conform if the group is especially friendly or close-knit, if the object the members are judging is ambiguous, or if the members have to make their opinions public. A group member is also likely to change his/her opinion if a large majority of the other members hold a contrary opinion to their own.

Carment (1961) found that the first people to vote publicly on a topic were the most confident of their opinions, with these first opinions often becoming the majority opinion. Other confident people then became group deviants who could either conform, change the group norms, remain a deviant, or leave the group. A group tended to reject deviants, however, if it could survive more effectively without them. Friendship amongst group members also increases conformity, although only if the group norm was set by the group itself and not an outside agent.

2.2.3.4 Roles

A role is a set of expectations which group members share concerning the behaviour of a person who occupies a given position in a group. An individual can hold both formal and informal roles in the same way they can be members of both formal and informal groups. Members with a high formal rank have more influence on group decisions than those with low rank, with Bales (1953) showing that a leadership role amongst a small group was given to the person who talked most and subsequently won most of the group decisions.

As mentioned, an individual may hold multiple roles in an organisation, all of which overlap in some way or another. Multiple roles are not easy to manage, and can result in *role conflict* and *role ambiguity*. There are four main types of role conflict (Fisher & Gitelson, 1983):

- **Intrasender role conflict**

  When you are told to do two or more things which are impossible (e.g. give a job top priority but do not neglect any other task)

- **Inter–sender role conflict**

  When two or more people in your role set are sent conflicting messages (e.g. sales want multiple versions of a product, production wants a single version).

- **Inter–role conflict**

  When a single person occupies multiple roles (e.g. conflicting interests between a department and an organisation).

- **Person–role conflict**

  When the person and the expected behaviours of the role do not coincide.

Role ambiguity occurs when individuals are unsure as to what to do when they occupy a role (Katz & Kahn, 1978). They may be unsure as to what is expected or what tasks are to be achieved, or unsure as to how to achieve the tasks. This can occur in environments facing rapid and far–reaching changes where new posts and promotions can lead to ambiguous roles being created.

2.2.3.5 Communication Networks

Groups are ineffective if the members have trouble communicating with one another. For small groups, there is generally little restriction as to who can communicate with whom. Large groups cannot have unrestricted communication, however, as it can become unwieldy and inefficient, if not impossible. As a result, many groups build up *communication networks* controlling the flow of information both up and down through organisational hierarchies. The form of communication network used by a group is important as it determines the way the group functions.

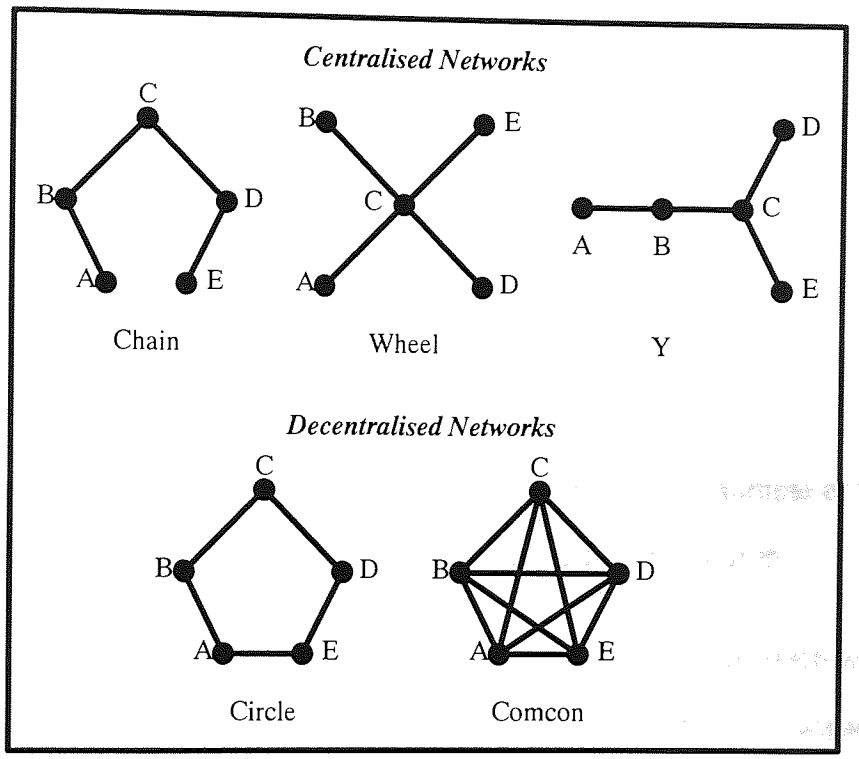

Figure 1: Communication Networks

Leavitt (1951) determined that a variety of communication networks are possible in a group or organisation: the *chain*; the *wheel*; the '*Y*'; and the *circle* network. This class was added to by Shaw (1978) who introduced the *comcon* network. The structure of these networks are shown in Figure 1 above.

In the chain network, information passes from one person to another, (i.e. A to B, B to C, C to D, D to E), resulting in a possible distortion of the message as it progresses. E is also unable to verify or negotiate the message with A, having to again send queries back through the communication chain if at all possible.

With the wheel network, all messages are routed through a centralised individual, C. C receives all the information and can send other information out to any of the peripheral members, with the peripheral members unable to communicate with one another. As a result, C controls the information flow in the network and is in a position of power, generally becoming the leader of the group. Leavitt found this form of network to be organised but unsatisfying to the group members.

The 'Y' network is similar to the wheel, except that information may pass in a chain–like fashion (e.g. A to B) by some peripheral members. The network is still centralised, however, with 'important' information still passing to and from the central member C.

The circle network is similar to the chain network except that information passes round to everybody, making any member both a sender and a receiver of information. Leavitt found that this network was active, leaderless, unorganised, and erratic, but was enjoyed by its members.

With the comcon network, each member can interact directly with any other member, requiring no intermediaries. The effect of this is the same as that for the circle network, except more pronounced. Small face–to–face peer group meetings are an example of this network, where each member has the ability of interacting equally in a discussion.

Both circle and comcon networks are decentralised. It is thus impossible to tell which member might become leader from their network positions – only the member's personality and skills will be the determining factors. Regarding group performance, research has found that centralised networks are best for simple tasks, with decentralised networks best for complex tasks (Forsyth, 1983). The communication network employed should also reinforce and reflect the status and role characteristics of the group, often serving to protect group leaders from being overloaded with requests and information, and allowing for the rapid and efficient transmission of news, goals, information and commands throughout the group (Baron et al., 1992).

2.3 Decision Making

Decision making is based around having a series of options and choosing which option to follow, and is one of the most frequent tasks required by an individual or group in any work

scenario. Decisions can range from choosing a slogan for an advertiser, to whether to promote a person for a manager, to what the content of a chapter should be for an author.

2.3.1 The Decision-Making Process

Before making a decision people go through a cognitive process from which a decision will result. Two main approaches have been developed to analyse this decision-making process, the *traditional, analytical model* and the *intuitive image theory.*

2.3.1.1 The Traditional, Analytical Model of Decision Making

Decision making is regarded as a series of steps that groups or individuals take to solve problems (Harrison, 1987). A general model developed by Wedley and Field (1984) shows eight steps in the decision-making process. Not all steps need be followed, however, but may be skipped or combined. The steps are:

1. *Identify the problem*

   Before solving a problem, it must first be recognised and identified. Cowan (1986) found that people often distort, omit, ignore or discount information relating to the existence of problems, generally as a result of feeling uncomfortable with a problem if it was recognised.

2. *Define objectives to be met in solving the problem*

   By setting objectives to be met, a range of possible solutions can be identified. Solutions can then be evaluated relative to the objective, with a good solution being one that meets the objective.

3. *Make a predecision*

   A predecision is a decision about *how* to make a decision. Depending on the problem, the style of the decision maker, and other aspects, managers may opt to make the decision themselves, delegate the decision to another, or have a group make the decision.

4. *Generate alternative solutions*

   Depending on the objectives to be met, various solutions to the problem need to be identified. People often look to historical choices in determining possible solutions, hoping they may provide ready–made answers (Stevenson et al., 1990). Brainstorming techniques (Osborn, 1957) may also be used to identify solutions.

5. *Evaluate alternative solutions*

   Solutions identified in step 4 may have different degrees of effectiveness in meeting the objectives from step 2. Some solutions, whilst being effective, may be difficult to implement, so additional aspects in a solution also need to be recognised.

6. *Make a choice*

   Choosing an acceptable solution is the step most thought about when one considers decision making. People often do not consider alternatives thoroughly, however, and do not choose the optimal solution.

7. *Implement choice*

   The chosen solution is implemented.

8. *Follow up the decision*

   Once a decision has been reached and a possible solution implemented, monitoring and feedback is needed to see if the problem still exists, or if implementation of the solution has led to any further problems. If a problem still exists, the decision–making process begins once again at step 1.

2.3.1.2 The Intuitive Image Theory to Decision Making

Whilst the traditional model of decision making holds for most decisions, there are certain situations and decisions where a more intuitive–based decision making is likely. Mitchell and Beach (1990) note that selecting the best alternative is not always a major concern when

people make a decision, but instead people choose that alternative which fits their own personal standards, plans, and goals. The researchers have developed an *image theory* (Beach & Mitchell, 1990) based on decisions concerned with either adopting or changing a certain course of action (e.g. should you adopt a new product or change an existing one?)

Image theory says that adoption decisions are made by performing a *compatibility test* followed by a *profitability test*. The compatibility test compares the action to be taken with various images, mainly individual principles, current goals and plans for the future. If there is any incompatibility with these images a rejection decision is made, otherwise the profitability test is carried out. Here, people consider which alternatives best fit their values, goals and plans, taking into account information from past experiences and other sources. This theory regards the decision–making process as both rapid and simple, being made using an intuitive process with minimal cognitive processing (e.g. doing an action because it "seemed like the right thing to do".)

2.3.2 Individual Decision Making

Individuals tend to believe they make the best possible decisions under particular circumstances, regarding themselves as "rational" beings. *Rational decisions* can be defined as decisions that maximise the attainment of goals, whether they be the goals of a person, a group, or an entire organisation (Linstone, 1984). This *rational–economic* view of decision making follows the traditional, analytical model, where an individual systematically searches through all available options for the optimum solution to a problem. The view also assumes that the decision maker has complete and perfect information, and can assimilate it in an accurate and unbiased manner.

Decision makers are human, however, and so the rational–economic view can be regarded as a prescriptive approach, one showing how decision makers should *ideally* behave in making their decisions. In reality, the limits of human cognitive ability lead to a series of short–cuts and simplifications in the decision–making process. For example, instead of examining *all* solutions and subsequently picking the *optimal* one, decision makers examine solutions until they reach one they feel is acceptable. These decisions are known as *satisficing decisions*, for

which an analogy is: "*making an optimal solution is like searching a haystack for the sharpest needle, a satisficing decision is like searching a haystack for a needle just sharp enough with which to sew*" (March & Simon, 1958).

Humans also lack the cognitive ability to make highly complex decisions in a completely objective way. This *bounded rationality* can manifest itself in many forms. Kahneman and Tversky (1984) categorise these cognitive biases using:

- ***Framing***

  People make different decisions depending on how the information is presented to them. Conservative (risk averse) decisions are made if a problem is framed in a manner emphasising the potential gains, whereas risk–seeking decisions are made if the same problem is framed in a manner emphasising the potential losses.

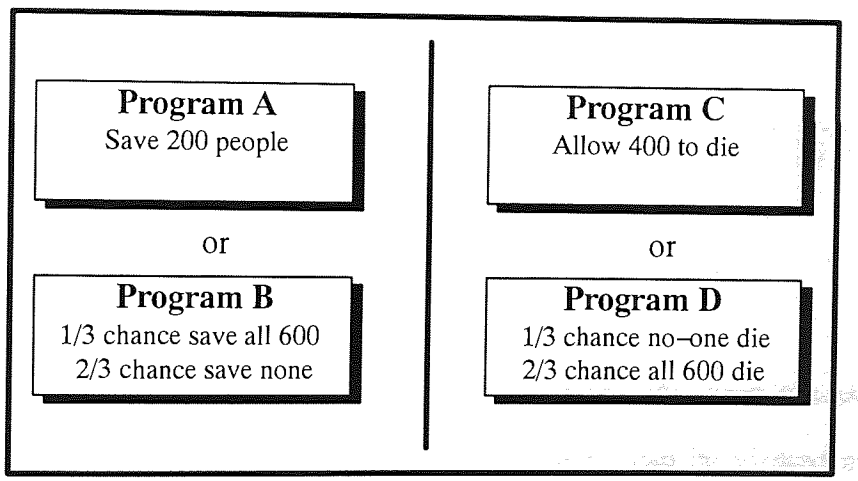

Figure 2: Kahneman & Tversky killer disease decision experiment

Figure 2 above shows the options Kahneman and Tversky used to prove that framing influenced decisions. People were first asked to choose between programs A or B, with 72% of people adopting for A and not taking a risk. The same people were then asked to choose between programs C or D, with 78% of people picking D and taking a risk. Programs A–C and B–D are identical, yet people make opposite choices depending on whether the options are framed positively or negatively.

- *Heuristics*

  People often try to simplify complex problems by using heuristics ("rules of thumb"). The *availability heuristic* refers to a tendency for people to base their judgements on information readily available to them, such as past experiences, which may be wholly inaccurate in the present situation (Dubé–Rioux & Russo, 1988). The *representativeness heuristic*, alternatively, refers to peoples' reliance on stereotyped perceptions, even when it is at odds with the base–rate information available.

- *Escalation of Commitment*

  Once a person makes a decision they become committed to it, needing to justify it to both themselves and others retrospectively. Once a decision has been seen to be ineffective, rationally the decision should be stopped or reversed. Staw (1981) found that, in reality, people often follow up ineffective decisions with further ineffective decisions in the hope of 'turning the problem around' and avoid admitting failure. It has been found, however, that an escalation of commitment can be avoided when the decision maker has limited resources, has overwhelming evidence of negative outcomes, or can diffuse responsibility for previous actions (Garland & Newport, 1991).

2.3.3 Group Decision Making

Groups are widely used in organisations, nearly always being involved in making the most important decisions in a company. Group decision making can be viewed as a *"process whereby groups move from initial disagreement to a sufficient level of agreement to satisfy some decision rule"* (Baron et al., 1992). The efficiency and quality of decisions made can be affected by a group's social structures, however, with groups having both their problems and advantages for solving problems and making decisions. To this end, groups need complex and varied techniques for managing the decision–making process.

2.3.3.1 Problems with Group Decision Making

The problems faced in individual decision making also occur in groups, and indeed may be increased depending on the group composition. For example, belief in either availability and

representativeness heuristics may be increased if all group members are from a similar class and background of people, or decreased if the members are from diverse situations. The main problems faced with group decision making are: group conformity, groupthink and group polarisation.

2.3.3.1.a Group Conformity

There are problems in decision making which are unique to groups, problems which tend to grow out of the social influence which can be exerted in groups. One of the major areas of social influence relates to how people tend to conform in groups, reaching consensus decisions even when those decisions are wrong (Asch, 1955). Research has found that conformity occurs most when:

- the issue is difficult, with conformity increasing as uncertainty increases (Deutsch & Gerard, 1955);
- there is unanimous group consensus, with isolated group deviants likely to be rejected and punished (Wilder & Allen, 1977);
- the group is admired, as rejection from friends is very threatening (Back, 1951);
- one's response is easily identifiable and subject to social rewards and punishments (Deutsch & Gerard, 1955); and
- one is scared, with fear increasing dependency needs by undermining confidence (Darley, 1966).

Group conformity has been found to occur least when:

- a person has great confidence in his/her expertise on the issue (Deutsch & Gerard, 1955);
- a person is strongly committed to his/her initial view (Deutsch & Gerard, 1955);
- a person has high social status (Harvey & Consalvi, 1960); and
- one does not like or respect the source of social influence (Hogg & Turner, 1987).

2.3.3.1.b Groupthink

Groupthink is *"a mode of thinking that people engage in when they are deeply involved in a cohesive in–group, when the members' strivings for unanimity override their motivation to realistically appraise alternative courses of action"* (Janis, 1972). Groupthink occurs when members do not want to cause dissent and agree with a decision even if having private reservations. The members then isolate themselves from outside influences and do not examine critically the decisions being made.

Janis (1982) noted that the major symptoms of groupthink are:

- an illusion of invulnerability (ignoring danger signs and taking extreme risks);
- collective rationalisation (discrediting warning signals contrary to group opinion);
- unquestioned morality (belief in the groups' morals and ethics);
- excessive negative stereotyping (regard opposing side negatively and ignore their views);
- strong conformity pressure (discouraging the expression of dissent);
- self–censorship (withholding dissenting ideas and information from the group);
- an illusion of unanimity; and
- self–appointed mindguards (protecting the group from negative, threatening information).

To overcome groupthink, Janis (1982) suggests four main measures:

1. Impartial leadership so that group members are not tempted simply to "follow the leader".
2. Each group member should be encouraged to air doubts and objectives (make it a norm).
3. Experts should be in attendance to raise doubts.
4. "Second chance" meetings should be held where members can express doubts about decisions made before they are implemented.

2.3.3.1.c Group Polarisation

Groups are often said to arrive at compromise decisions. In reality, however, research has shown how groups made *riskier* decisions than individuals (Stoner, 1961), and that groups tend to take more extreme decisions than the initial preferences of the group members (Myers & Lamm, 1976). That is, the individual members' views become more extreme (or polarised) during group discussions and, especially if the group tends to conform or the groupthink phenomenon is occurring, can cause more potentially disastrous decisions. This is especially true for homogeneous groups, where the members are more likely to hold the same views and opinions (Dyson et al., 1976).

2.3.3.2 Advantages of Group Decision Making

Groups have a number of advantages over individuals in decision making. First, with complex decisions, groups are more able to use specialisation so that each member's expertise can be used to gauge the various advantages and disadvantages of each solution. This decreases the cognitive load on each individual. To be effective, however, the group needs to consist of varied group members with complementary skills. Also, the problems with group decision making need to be removed, with each expert being able to air their views without risk of intimidation (Wanous & Youtz, 1986).

Michaelsen et al. (1989) found that groups also performed better on complex tasks compared to the best individual in the group. That is, the advantages of group work is more than just the simple summation of individual member's skills, but has added synergy created when a group of people help each other and create a climate for success.

Finally, groups of people can overcome some of the deficiencies faced in individual decision making with respect to the availability and representativeness heuristics. If a group has varied members, each member will bring their own experiences into the discussion which should provide a more rounded, if not more accurate, account of the area being discussed. A group which conforms well, and who are collectively responsible for a decision, may have a larger problem with the escalation of commitment phenomenon

2.3.3.3 Group Decision Making Techniques

A range of techniques have been developed to aid groups of people in the decision making process. The techniques developed fall into two categories, structuring the group discussion in a special way, and improving the skills individuals may bring to the decision–making situation.

2.3.3.3.a Group Discussion Structuring

Figure 3 below illustrates the main techniques which are used in structuring group meetings.

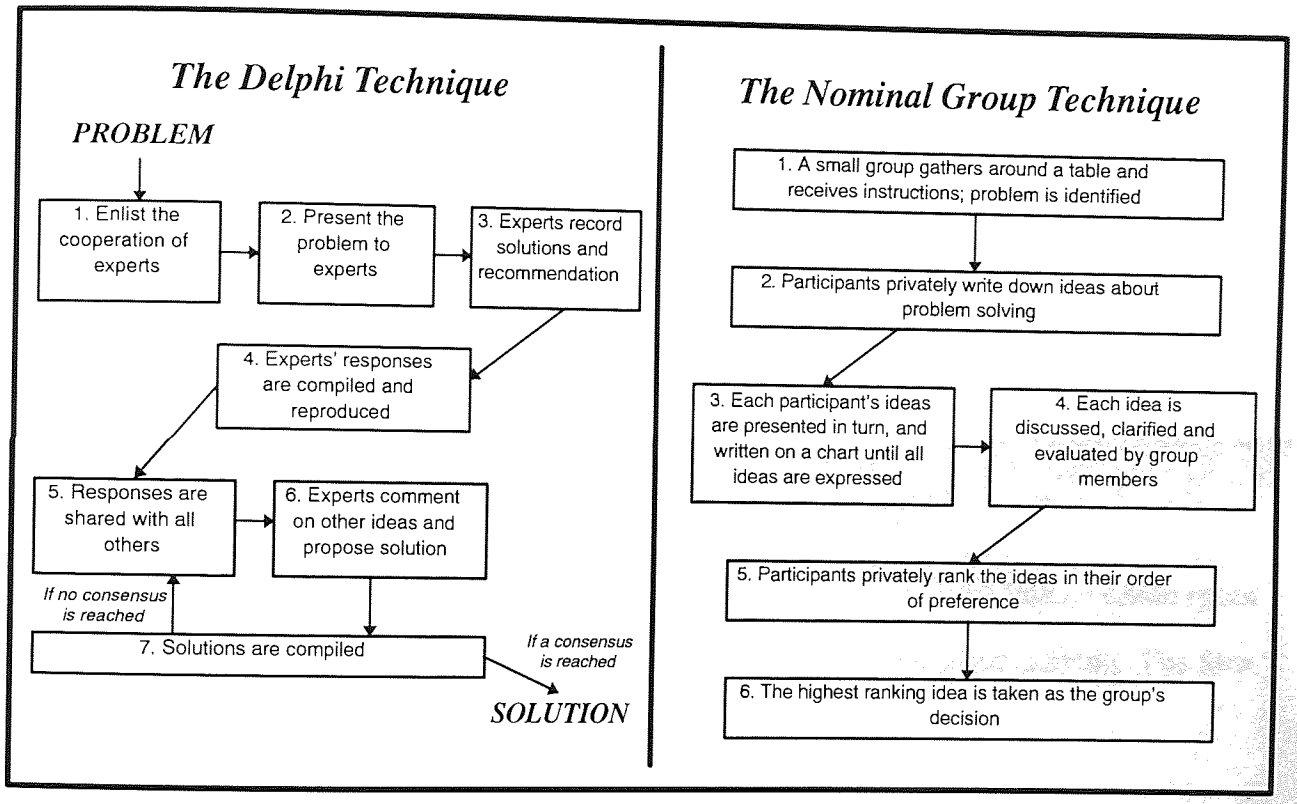

Figure 3: Techniques to improve group decision making.

Adapted from: J. Greenberg & R.A. Baron Eds. (1993) *Behavior in Organizations (4th Edition)* pp. 563–564.

The Delphi technique, developed by the Rand Corporation (Dalkey, 1969), uses the help and guidance of expert opinion in the creation and evaluation of alternative solutions to a decision making problem. The technique comprises a way of systematically collecting and organising the opinions of several experts into a single decision, generally through the mail service. It is an iterative approach of asking experts for individual solutions, distributing the returned solutions to each expert, getting comments on them, distributing the comments, and

continuing through this process until a consensus solution is reached. The technique's value is that many experts can make a decision without the time and cost of meeting at one place, while its main drawbacks are the time taken to make a decision and the possible lack of commitment to a decision which has not been made during a face–to–face meeting.

The Nominal Group Technique (NGT) is a technique used when only a short time is available for making a decision. In this technique, a small number of individuals are brought together who consecutively present personal solutions to a problem and share their reactions to each others' contributed solutions (Gustafson et al., 1973). The people do not form a 'group' as such, in that they do not need to agree unanimously on a decision for it to be accepted, but instead choose a decision solution by using a voting system. The voting system used consists of each member privately ranking the solutions in order of individual preference, with the solution with the highest rank taken as the group's decision. Advantages of the NGT include rapid decision–making and member satisfaction, whereas disadvantages are the need for a trained group leader and, due to its voting system, the fact that not all members may agree with the decision being made (although the majority will).

2.3.3.3.b Improving an Individual's Decision–making Skills

Bottger and Yetton (1987) found that if individuals were trained to avoid four common types of error then errors would be greatly reduced in creative group problem solving. The four problems to be aware of (and avoid) are:

1. Hypervigilance

   Groups should work through each potential solution thoroughly, reassuring individuals that they can solve the problem. This avoids people searching frantically for quick solutions out of desperation when one solution isn't working.

2. Unconflicted adherence

   Instead of focusing on the first potential solution, the decision maker should think about difficulties associated with it, force him/herself to consider other ideas, and note the unique characteristics of the problem at hand compared to previous problems.

3. Unconflicted change

   Instead of quickly changing his/her mind to a new idea, the decision maker should consider the risks and problems with the new solution, note the good points of the first solution, and compare the relative strengths and weaknesses of both ideas.

4. Defensive avoidance

   Instead of avoiding a difficult problem, the decision maker should set some time to keep examining the problem for a solution, avoid disowning responsibility (regarding the problem as unimportant), and not ignore potentially corrective information so as to be done with the problem.

2.4 Group Cooperation or Conflict

In organisations, individuals and groups can either work for or against one another. Greenberg and Baron (1993) note that there are three major social processes which can occur between individuals or groups:

- Prosocial behaviour, consisting of actions by groups/individuals which assist others with no requirement that the recipients return the favour;
- Cooperation, where groups/individuals work together towards some mutual goal; and
- Conflict, where actions by an individual/group are perceived by others to have negative effects on their interests.

Of these processes, *cooperation* and *conflict* are the processes most important to successful group management.

2.4.1 Group Cooperation

In group cooperation, two or more individuals or groups work together to enhance progress towards some shared goal. Cooperation does not develop, however, if the goals sought by the

individuals or groups cannot be shared, such as with two companies bidding for the same contract. In these cases, *competition* often develops, where each side strides to maximise its individual gains, often at the expense of others (Tjosvold, 1986).

Cooperation occurs depending on how individuals perceive or react to those they would be cooperating or competing with. Generally, people follow the principle of *reciprocity* – to behave towards others as they have acted towards you. Thus, if somebody acts competitively, one generally responds with competition, and similarly for cooperation. Open *communication channels* for exchanges of views may increase interpersonal trust among members. If closed, however, cooperation may fail if individuals or groups are not aware of what one another are doing. The *individual personalities and perspectives* of people also play a major role, with research by Knight and Dubro (1984) showing that the main categories of personality are:

- Competitors, whose prime motive is to do better than others;
- Individualists, who want to do as well as possible with little regard to others;
- Cooperators, who want to maximise joint outcomes; and
- Equalisers, who want to minimise differences between them and others.

Comparing cooperation and competition, Deutsch (1949) found that cooperating groups had fewer communications difficulties than competing groups, with cooperation increasing group productivity. Church (1962) further found that when group members were motivated to cooperate, they showed more positive responses to each other, were more favourable in their perceptions, were more involved in the task at hand, and had greater satisfaction with the task. Cooperation inside a group also tended to increase if the group was competing against another group.

2.4.2 Group Conflict

Conflict is common in interactions between individuals, and between groups of individuals, and refers primarily to instances in which groups or individuals in an organisation work

against rather than with one another (Thomas, 1992). Conflict is commonplace in organisations, with managers spending approximately 20% of their time dealing with conflict and its impact (Thomas & Schmidt, 1976). It is thus an important aspect in the development and life of a group, either through conflict internal to a group, or conflict between groups.

Conflict can be described as a trade–off between concern with one's own outcomes against concern with the outcomes of others. Five basic styles of handling conflict can be seen from this trade–off: *competing*, *collaborating*, *avoiding*, *accommodating*, and *compromising* (Thomas, 1976). Figure 4 below shows this trade–off graphically.

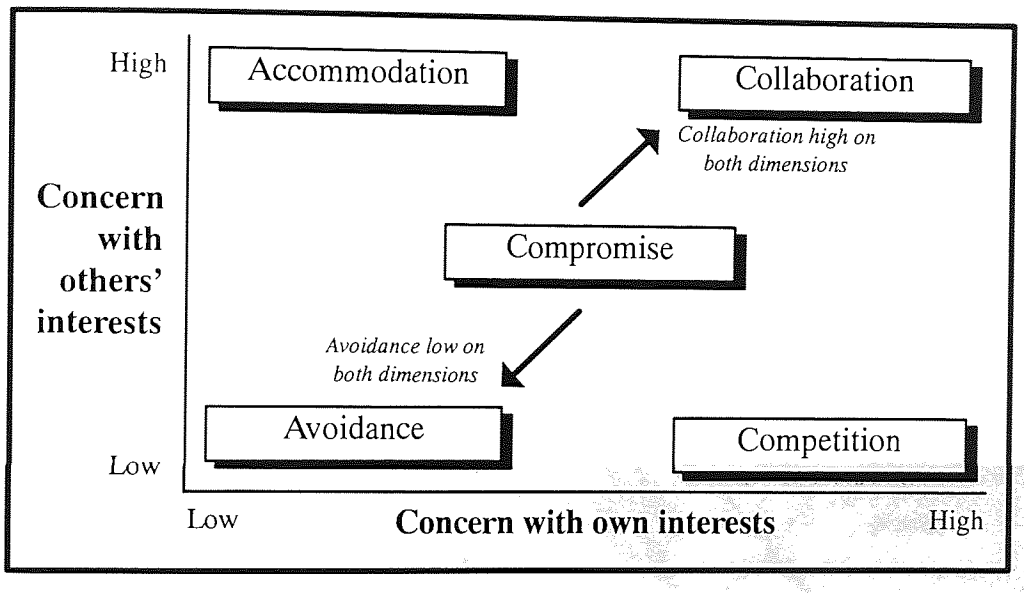

Figure 4: Basic styles of resolving conflicts.

A *competitive style* has one participant seeking to dominate the process, without regard for others, and may be useful for quick decisive action or where unpopular decisions are perceived as necessary.

A *collaborative style* has the participants seeking to understand their differences and achieve a mutually beneficial solution, and may be useful where participants' insights and commitment are important.

An *avoidant style* is used when the conflict is recognised but suppressed by one or more parties, or handled by withdrawal, and is useful where the conflicting issue is unimportant or where information gathering is key.

An *accommodative style* is used when one party places the others' interests above their own, and is useful when the issue is far more important for one party, when one party wishes to minimise loss, or where there is a desire to build harmony.

A *compromise style* has both parties making concessions to reach a compromise, and is useful when temporary solutions or expedient solutions are needed, especially under a time pressure or where both goals are directly opposed.

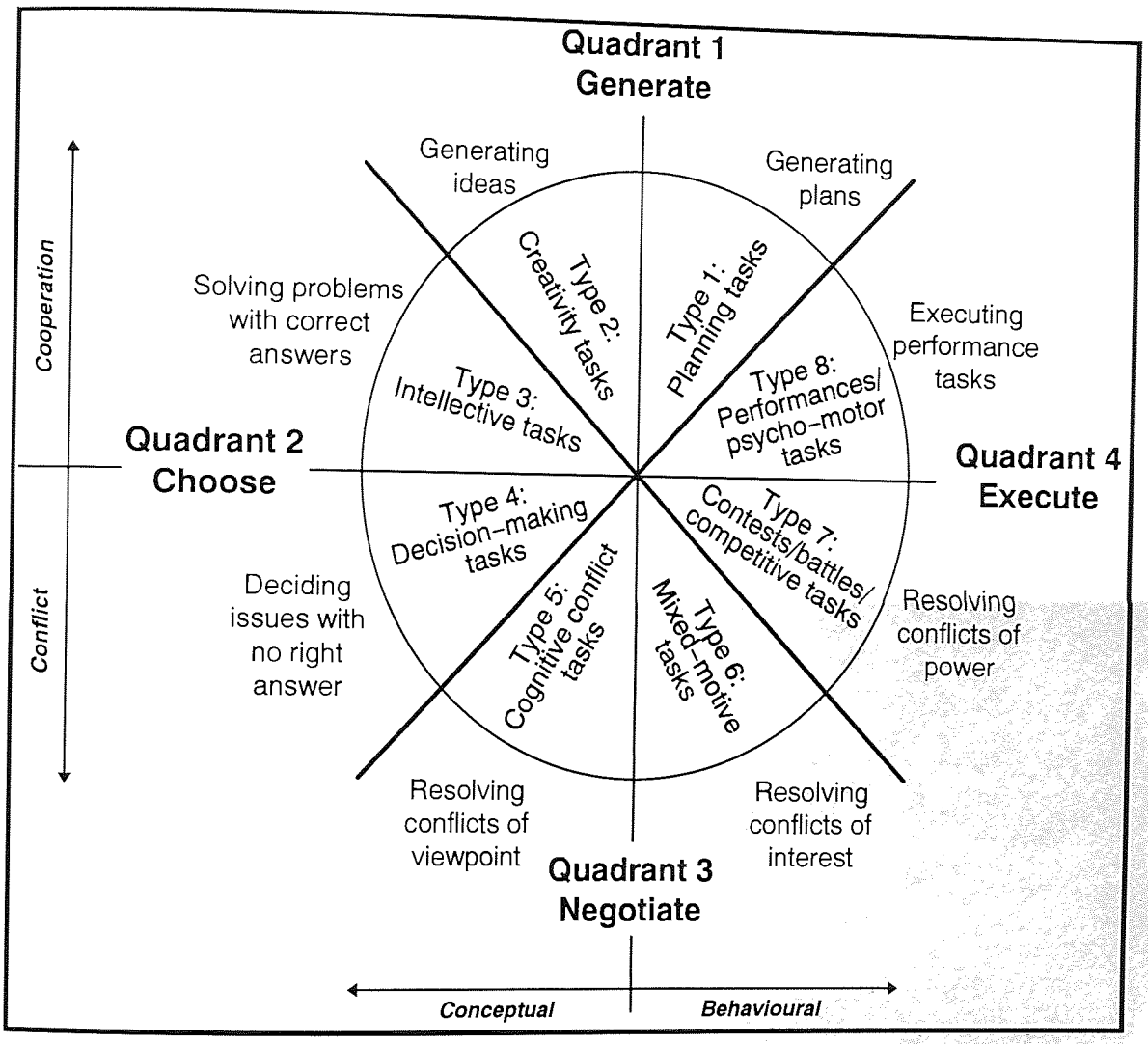

Figure 5: The "Group Task Circumplex"

Adapted from: McGrath (1984). "Groups: Interaction and Performance". Prentice-Hall,Englewood Cliffs,NJ

McGrath (1984) has categorised types of group behaviour under four categories: *generating*, *choosing*, *executing* and *negotiating*, for which the actions of choice, execution and negotiation have a key role in determining the type of conflict. For example, deciding on an issue with no 'correct' answer (a typical decision-making task) involves a conflict concerning the choice made and the negotiations involved in making that choice.

Alternatively, generating ideas (as in a typical brainstorming task) relates more to cooperation in generation and choice of ideas. Figure 5 above depicts these major group activities, and the level of conflict or cooperation they are likely to exhibit.

As can be seen, conflict occurs to some degree in most of the tasks performed by groups, and especially in negotiations. Conflict, however, may not be seen to be undesirable, with Robbins (1974) observing that conflict had a useful role in organisations as it provided stimuli to innovation, was a major weapon against stagnation and resistance to change, and by questioning and evaluating the received wisdom it helped to reduce problems associated with groupthink.

2.4.3 Assertions and Assumptions about Conflict

Easterbrook et al. (1992) have put forward a series of assertions about conflict based on their review of the literature, and have then attempted to show whether the assertions are valid. The assertions they have made are:

1. **Conflict is inevitable.**

   Dahrendorf (1959) regards conflict as endemic in society, as conflict is based around a clash of ideologies. Whilst conflict may not be *inevitable* for every possible group, the tensions which lead to conflict are apparent in most groups and for some groups, such as group psychotherapy, conflict *is* inevitable (Unger, 1990).

2. **The more cohesive the group, the less conflict there is.**

   Lack of cohesion in a group was found to be the greatest cause of conflict in 70% of naturally–occurring groups surveyed (Weinberg et al., 1981). This cohesion can be taken too far, however, as witnessed in the groupthink phenomenon, with Gero (1985) advocating the importance of disagreement to the outcome of group decisions. The measure of cohesiveness in a group can also depend on its composition, with Collaros and Anderson (1969) finding that heterogeneous teams experienced more conflict at first in their interaction processes compared with homogeneous teams. As stated above, however, homogeneous groups are more likely to suffer from group polarisation.

3. **Occurrence of conflict varies with the development of the group.**

As described previously, groups pass through a number of stages in their development (Tuckman & Jensen, 1977). During the storming and norming stages of group development, conflict occurs most often. If a group does not reach or misses some stages in its development, however, conflict may also occur.

4. **The more communication there is between people, the more opportunities there are for conflict.**

Much research has been carried out on communication networks and group performance, such as by Leavitt (1951) and Shaw (1978) described previously. As well as the type of network used, McGrath (1984) has also found that the narrower the communications bandwidth used, the more task–focused the interaction between the members becomes. As there are no non–verbal cues, interpersonal and social aspects are not conveyed. Thus a rich communication medium can produce "noise" that distracts from the task, yet aids for the development of interpersonal relationships and can permit group members to exercise regulatory functions in their interaction and achieve a better success rate in conflict resolution by not using high–risk conflict strategies such as bluffing (Crott et al., 1980).

Thomas (1976) has observed that a decrease in communication may intensify a conflict, such as when one party uses communication to manipulate or control another party. This allows opposing parties to maintain distorted stereotyped views of one another, and acts to feed their hostility to one another (such as armies preventing their soldiers from fraternizing with the enemy).

5. **Clearly defined roles reduce conflict.**

As discussed previously, roles given to members can cause conflict depending on the role and task they are expected to perform. A lack of any clearly defined roles can also lead to conflict, with Wood (1989) arguing that the lack of role assignments increases the probability that a task group will evolve into an informal group which does not complete the task assigned to it.

6. **Anonymity and physical separation contribute to conflict.**

The use of computer-mediated communication (CMC) can change the pattern of communication, the distribution of information and the nature of interactions between people (Sproull & Kiesler, 1986). CMC also led to the possibility of group activities occurring over geographically distributed locations and with anonymous use.

This anonymity has been found to have both positive and negative effects in groups. Jessup et al. (1990) note that it may detach an individual from their comments and lead to a reduction in normal restraints on behaviour (a de-individuation effect), with Kiesler et al. (1984) adding that CMC failed to provide individuating details about people which may be provided by their dress, location, demeanour, and expressiveness. Sproull and Kiesler (1986) argue, on the other hand, that CMC provides increased social communication, uninhibited behaviour allowing new ideas to flow, and the reduction of geographic, temporal, departmental and status barriers.

7. **Conflict can be productive.**

While some conflicts can have negative effects (such as in power struggles or personal antagonisms), conflicts can also have very positive effects. Deutsch (1969) suggests that conflict prevents stagnation, stimulates interest and curiosity, is the medium for airing problems and arriving at solutions, is the root of personal and social change, and leads to the "arousal of the optimal level of motivation".

Thus, the effective management of conflict (Robbins, 1989) is the major consideration relating to conflict, rather than its avoidance or elimination. Conflict, effectively managed, is a necessary precondition for creativity (Hall, 1971).

8. **Actively eliciting conflicting viewpoints can reduce groupthink.**

The groupthink phenomenon has been described previously, and occurs in highly cohesive groups which are isolated from experts and where the members feel a sense of invulnerability. Recommendations by Janis (1972) to reduce groupthink can be regarded as ways to positively sanction conflict as both desirable and necessary (Gero, 1985). She

notes that while consensus is desirable at the end of a decision–making session, it is counter–productive to try and reach a consensus from the start as alternatives will not be properly considered and the decision quality will suffer.

9. Conflict has to be resolved for parties to continue to work together.

Depending on the nature of the conflict, resolution of the conflict may have a more negative effect than the effect of the original conflict. Baxter (1982) notes, however, that avoiding conflicts may result in a stockpile ending in a "super–conflict" of all unresolved issues. Also, as stated above, some conflicts are seen as positive, and so it is the management of the conflict which is important – parties may continue to work together with a conflict which, while unresolved, is still being managed.

10. Size of a group affects the occurrence and resolution of conflict.

Conflict occurs more in smaller groups due to the fact that each member is more engaged in the task. This conflict may not be destructive, however, as larger groups have problems with redundant extra members and complex communication patterns (Brooks, 1975). As mentioned previously for small groups, Bales and Borgatta (1955) note that the number of members is key in conflicts, with five members being the optimum number for conflict resolution.

11. Personality has little effect on the development of conflict.

Research has shown that individual personality traits have very little impact on the *style* of conflict strategy adopted (Putnam & Poole, 1987). Personality may, however, influence the initial behaviour of participants. In absence of other constraints, the participants' approach to negotiation is determined by their personality. Once a negotiation is underway, the personality of the other person is far more important, as the course of the negotiation will depend on how the participants react to each other (Hermann & Kogan, 1977).

12. A strong leader is needed to resolve conflict.

The selection of a leader might itself lead to conflict if the person does not hold the appropriate esteem of the group, or does not perform as the leader is expected to do so by the group (Shaw & Harkey, 1976). A strong leader is often categorised according to their decision making style, ranging from democratic to autocratic. Depending on the conflict the effectiveness of any particular style will vary, with an autocratic leader suppressing a conflict and a democratic leader prolonging a conflict (Howell et al., 1986).

13. Conflicts are unlikely to be resolved if participants argue from entrenched positions.

Pace (1990) describes conflicts where the opposing parties have entrenched views as "competitive conflict", characterised by defensiveness, hostility and escalation, compared with "cooperative conflict", which is positive, supportive and peace-keeping in nature. DeBono (1985) also notes that the best way to resolve a conflict is to reformulate the problem, which is best achieved if the group are separated from their positions. Subsequently, a confrontational approach to the problem solving process can be costly, while a collaborative approach is mutually awarding.

14. Articulating conflict helps in its resolution.

Conflicts that are not articulated may accumulate to produce breakdowns in group interaction (Baxter, 1982). Therefore by making conflicts explicit, understanding should be enhanced and group members focused on the same set of issues. Pace (1990) also notes that the articulation of a conflict is used as a prelude to its resolution. In certain cases, however, conflicts which cannot be resolved should not be articulated, with suppression being used to avoid senseless confrontation.

15. There is a positive relationship between levels of participation and satisfaction.

Member satisfaction in a group increases if the members are able to communicate their ideas without fear of punishment (Thomas, 1976), with members less satisfied in larger groups as the possibility of active participation decreases. Pood (1980) has distinguished two main types of satisfaction: *social satisfaction*, the satisfaction with the interactions of

the group; and *decision satisfaction*, the satisfaction with the decision resolution. While conflict management techniques can improve social satisfaction, decision satisfaction does not increase as the conflict is never really resolved as far as the individual is concerned.

Kimberly (1987) notes that satisfaction depends on the role or task given to the participant. If the person's skill–level is too high for the role then boredom results, while if the person's skill–level is too low for the role then the person cannot perform adequately in the position.

Falk (1981) further discusses the use of unanimity/majority rule in decision making, finding that unanimity rules maintain equality in groups which are already equal, while majority rules help equalise the distribution of power among group members, facilitating discussion and leading to higher quality solutions and greater satisfaction.

16. The "loser" in a conflict will try to save face, and the "victor" may help the loser to do this.

In negotiations, the issue of "saving face" may be as important, if not more so, than the original conflicting subject. Sermat (1964) has found that face–saving occurs, through trivialising the subject, by claiming no concessions have been made, or by stressing the importance of reaching an agreement. Negotiated solutions therefore often have face–saving elements built into them, thus making the compromise or capitulation more palatable. Swingle (1970) found, however, that having anonymity, or having the belief that you will never meet your counterpart, can reduce the need to save face.

These assertions and assumptions are all important elements to take into consideration when developing systems to help negotiation and conflict resolution.

2.5 Computer support for Groups

This chapter has to date looked at many of the processes and issues involved in all aspects of group working. Computer systems have been developed to aid group members in their work,

collectively known under the name of "*groupware*". These are tools and applications which have been developed for the purpose of aiding groups work or function together more productively. In its loosest sense, a telephone can be regarded as a groupware system as it enables two (or more) people to communicate (and thus work more effectively) together.

Kraemer and King (1988), Nunamaker et al. (1991), Rodden (1991) and Easterbrook et al. (1993) have each surveyed and reviewed current groupware systems. This section describes some of the main categories of groupware system, with examples of their use. Tools designed for collaborative writing will be discussed in the following chapter.

2.5.1 Messaging Systems

Messaging systems are groupware tools aimed at improving the communications medium between participants, often situated in geographically dispersed locations. Electronic mail is the most widely used messaging system, being asynchronous in nature, having fast delivery times, and containing contextual information in its header fields (e.g. author, subject, distribution, date).

Computer conferencing systems are based on electronic mail systems, with the addition that a structure is imposed on groupings of messages and that the messages are generally held at a central database. Thus, instead of *sending* a message to an individual or group as in electronic mail, a user *posts* a message into a conference that is readable by any member of the conference. Conference systems may also be synchronous, with users posting messages on a shared working space and with the inclusion of audio and video channels leading to teleconferencing and videoconferencing systems (Sarin & Greif, 1985; Ahuja et al., 1988).

2.5.2 Argumentation Systems

Argumentation systems are systems built on a model of cooperation, conflict, goals and issues. These systems are often categorised through their use of hypertext technology for structuring an issue to be resolved. They resemble nets of connected nodes, which may consist

of text, graphics, or some other media (such as sound), and links between the nodes denoting associations between the nodes.

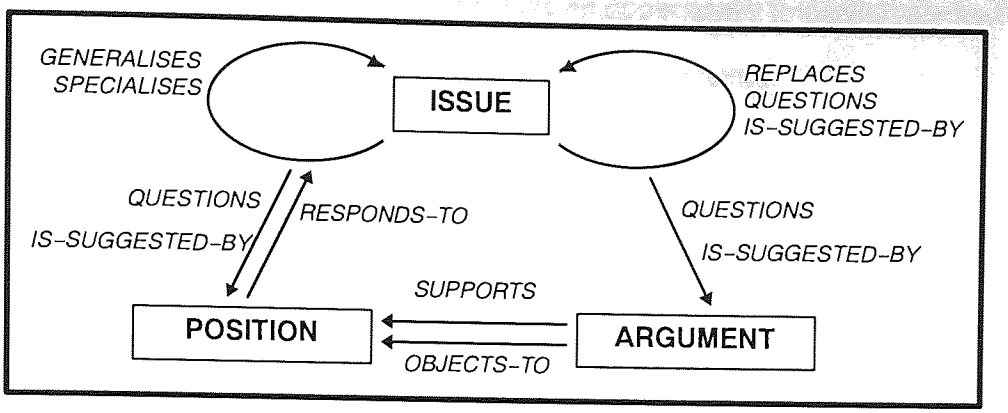

Figure 6: The set of rhetorical moves available in gIBIS.

An example of an argumentation system is gIBIS (Conklin & Begeman, 1988), a tool based on a model of design deliberation called Issue Based Information System (Kunz & Rittel, 1970). In this model, a structure is imposed so that participants can only respond with reference to issues, positions and arguments (as in figure 6 above).

Another example is Cognoter (Stefik et al., 1987), a tool providing an aid to brainstorming, organising and evaluating ideas. With this system users can prepare notes privately and then publish them onto a central liveboard where they may be discussed and argued about.

2.5.3 Group Decision Support Systems

Group decision support systems (GDSSs) include those systems mentioned in the categories above. That is, an argumentation system may also be regarded as a GDSS. More specifically, however, GDSSs often include tools such as voting systems and structured decision aids, where alternatives can be split up, ranked, and then voted on.

GDSSs are generally formed using specially-designed *decision rooms*, such as the Colab project at Xerox (Stefik et al., 1987) or the Collaborative Management Room at the University of Arizona (Nunamaker et al., 1991). These rooms were designed according to ergonomic principles, such as sunken terminals to allow for eye contact in the room, and the use of a *facilitator*, a person who can lead the group participants through their usage of the technology

in order to arise at an outcome for the meeting. This facilitating role could be purely technical, as a software and network trouble-shooter, or more sociological, with the facilitator managing the meeting to ensure all members have an opportunity to participate and that the decision processes and procedures required are followed correctly.

2.6 Conclusion

This chapter has looked at the area of cooperative working. Starting from an examination of the group – how it is created and what its characteristics are, to how it communicates – the chapter has then focused on the decision making process, both from an individual and a group perspective. Research into cooperation and conflict was then presented, finishing with a set of assumptions put forward with regard to conflict. Finally, the chapter has described a number of computer systems which have been developed to help groups communicate and work together more effectively.

The next chapter will shift focus from the collaborative aspects of the group to look more specifically at the areas of writing and text processing, concluding with an examination of existing collaborative writing systems.

Chapter 3

Text and the Writing Process

3.1 Introduction

Writing can be regarded as one of the most prevalent forms of activity performed by humans, with the written word forming a cornerstone of modern society. In the past, both printing and writing have been deemed to be specialist crafts for learned men. Technology, however, has made the opportunity to write and publish increasingly accessible to a wide range of people, stemming from the Guttenberg Press in the 15th Century, where mass printing became possible, to desktop publishing on personal computers, where printing has become both cheap, quick, and easily accessible.

Whilst *printing* has become easier, *writing* still remains a complex creative task. For a collaborative writing tool to be used, it needs to have facilities for text processing and aids for authors writing both in close collaboration or separately apart.

There is a large body of research on printing, writing, and text processing. In the context of this thesis it is impossible to provide a complete review of this research, so instead, some of the main aspects of the field will be examined. To this end, chapter 3 will first look at some of the issues to be addressed for both text and text processing, focusing on many of the typographical and structural aspects of text. It will then look at the process of writing by individuals, and computer tools available for aiding this task, concluding with the writing process for groups, and how computer systems may be used to aid authors in their work.

3.2 Text, Text Design and Text Processing

According to Winter (1977), the structure of text plays a primary role in the comprehension of all meaningful, informative prose.

In humans, knowledge is organised in memory through networks of interrelated representations of objects, with *schema theory* the most widely accepted model to describe this knowledge organisation (Rummelhart & Ortony, 1977). Schemata are used to represent knowledge (remembering) and interpret incoming information (understanding) based on previous knowledge. Words and word combinations trigger different schemata and, unless the words form some kind of structure, knowledge relevant to a new situation is difficult to retrieve and new information is not related meaningfully to past experience.

Text and discourse needs to be implicitly organised in order to be meaningful, with both the structure and content of text playing an important role in memory (Thorndyke, 1977). Text may mimic schemata by consisting of embedded hierarchical idea units, having both a micro–structure (the meanings of individual words and sentences) and a macro–structure (the "gist" of the text) at any number of levels (Kintsch & van Dijk, 1978). Thus the stronger the organisation of a text, the more likely it will be assimilated by the reader.

This section will look at how text may be organised. It will begin with a model for organising thoughts and ideas for a piece of work, then show how structure is an important aspect of documents, and then note how a piece of text can be created with this structure, both implicitly in the content and explicitly through typography. Finally, it will look at how typography can be used for different purposes in a text.

3.2.1 First Thoughts: Determining the Structure of a Text

As mentioned above, a text needs some form of structure in order to be more easily assimilated by the reader. When starting a writing project, many authors adopt a linear method of determining what they should talk about and how they should talk about it. This process is generally aided through the use of outliners (such as in Microsoft Word™) or through paper plans and lists. These lists are generally the end result of several other lists, which have been altered and changed to finally produce a final guide to work from. Fields (1982) notes that when determining the basic content of a text, authors are simultaneously trying to solve three interrelated problems if they try to use a linear outliner. These problems are:

- deciding the presentational viewpoint, that is, determining who the readers will be, what they will want to gain from the text, and what prior knowledge they can be expected to have;
- naming the key issues of the text; and
- ordering the key issues in a relational framework based on the presentational viewpoint.

Most people find some difficulty in solving this type of simultaneous problem, which accounts for the alterations and changes which need to be made to a list before it is complete. To counter the problems of viewpoint, key issues, and sequence, Fields suggests the use of *pattern notes*.

Pattern notes, also known as *mind maps*, (Buzan, 1983), are a means of reviewing alternatives as we think of other ideas to be included, exploiting the fact that our mind jumps from one point to another in a non–linear manner as we think about a topic. This causes the material to be organised in a relational, as opposed to a linear manner. It also means that only the key issues are initially addressed, allowing the presentational viewpoint and issue order to be decided afterwards. As the notes are in a relational order, different linear presentations may also be considered for different audiences, thus increasing the versatility of the notes structure.

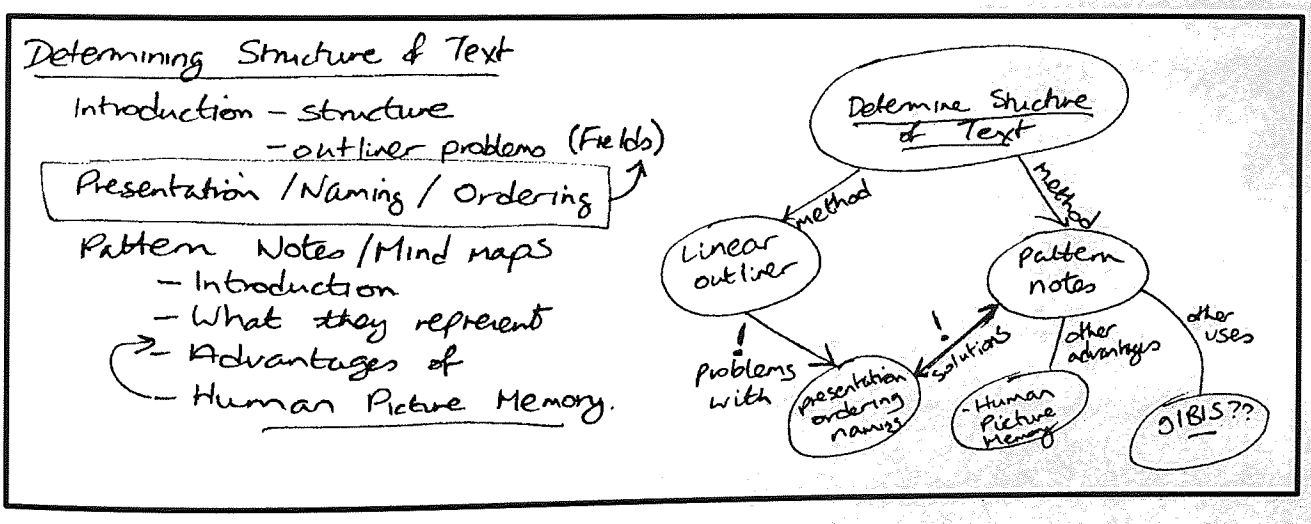

Figure 7: A linear representation and a pattern note representation for part of this chapter.

Figure 7 above shows an example of a linear outline and a pattern note for this section. From this we can see other advantages for the use of pattern notes. First, the whole subject is

summarised on one page, with key issues in the centre and those of least importance around the edges. Second, the notes can show inter–linkages of complex issues in a relational manner, such as where two sub–topics link together away from the central issues. Finally, as mentioned previously, the notes have separated the problem of choosing the issues to be dealt with from that of listing them in a linear sequence.

Pattern notes do not deal with organising the actual content body of text, but simply which key issues it should relate to. This relates to the structure of a document, with a pattern note translated into a hierarchy of issues starting from the key issues at the centre of the note, which finally results in a linear form which can be used as the basis of a document.

Pattern notes may also be used for other purposes, as in the development of a conceptual network. This is a way of visualising the body of knowledge about a particular subject and which can be viewed from a number of different positions, with each observer giving a different emphasis on different facets of the topic (Lewis, 1974). This could be used, for example, in a negotiation scenario where each participant has different concerns about what is being negotiated.

Finally, Haber (1970) and Franken & Rowland (1979) found that human picture memory is very efficient for retaining information, with humans having a large storage capacity for pictorial information. As pattern notes are a very visual system, they can be used as a form of memory aid compared to outline lists which are learnt off rote. These pattern notes may also be used as an advance organiser, a summary, or both.

3.2.2 Document Structure

As shown above, pattern notes can be used as a template for building the main elements which are required in a document. A document, however, does not consist solely of blocks of text addressing each issue detailed by the pattern notes. Instead, a document forms a highly hierarchical logical structure, generally consisting of some front matter, a main body, and some back matter, with the issues created from the pattern notes forming the main body (Coombs et al., 1987).

The complexity of these structures can vary greatly from those required for a simple memo to that for a complex textbook. Many elements are reusable, however, with some of the elements which are incorporated in a memo also able to be used in a textbook (Bryan, 1989). This is particularly relevant for documents which have different uses but similar structure and content. For example, a single document may form the basis for a chapter in a book, a journal article, a presentation, and a set of world wide web pages. Instead of creating multiple instances of the document, careful use of elements in a document may reduce repeated work and increase the consistency of the work.

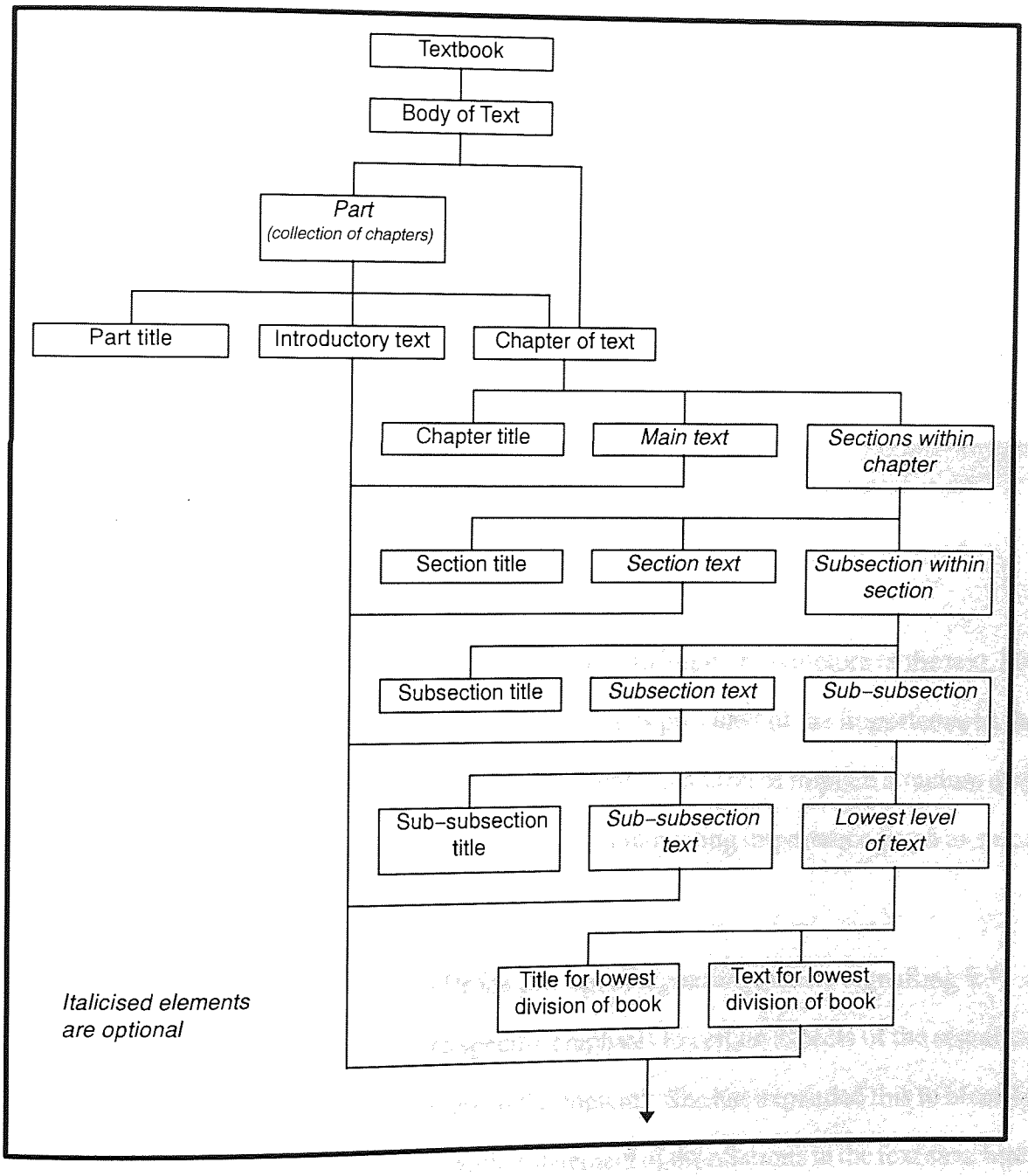

Figure 8: The structure of the main text in a typical textbook.

Adapted from: M.Bryan (1989) *SGML:An Author's Guide to the Standard Generalized Markup Language* p51

Figure 8 above shows an example of the structure used in the main text of a typical textbook. From it can be seen that the main text is simply a hierarchical structure of sections and subsections, with some front matter optionally at the beginning. This hierarchy can be expanded further to identify the main text elements in each block. For example, a block of text could consist of a series of paragraphs, a set of topics, lists, figures, and so on. A figure could in turn consist of a picture and a caption, while a paragraph could consist of highlighted phrases, quotations, references, and lists, as well as the standard text.

Thus, as well as having an extensive outward structure logically and typographically, the text of a document (the individual paragraphs) does not form a flat structure in which all ideas are equally important, but is instead also a highly hierarchical structure where certain elements are more central than other, supportive or subordinate elements (Duchastel, 1982).

3.2.3 Adding Structure to Text

Once a basic structure for a writing project has been determined, that structure needs to be reflected in the prose written. To do this, a piece of text may be given its structure either implicitly, explicitly, or both.

3.2.3.1 Implicit Structure

Implicit structure concerns the use of words and wording to imply the structure of the text. For example, by repeating arguments in a text, an indication is provided of the importance to the overall theme of the passage that the arguments represent. This form of implied structure may be used, for example, in preference to an explicit way of indicating importance (such as using bold, italic, or underlining).

Meyer (1975) has studied discourse under the concept of *signalling*, where signalling is "...a non–content aspect of prose which gives specific emphasis to certain aspects of the semantic content or points out aspects of the structure of the content". She has expanded this to identify several types of signalling to include "...explicit statement of the relations in the text structure, preview statements, summary statements, and pointer words or evaluative signalling" (Meyer, 1980).

3.2.3.2 Explicit Structure

Explicit structure is the use of typographical techniques to signal the content structure of a text in very explicit ways. This could be through blocking text, italicised type, diagrams, spacing, or many other typographical techniques. These techniques are used to *focus* the attention of the reader, and in this guise are used primarily during the perception and processing of information, rather than during retrieval. As well as focusing the reader, Tulving and Thompson (1973) have found that these typographical cues are encoded into memory along with the content of prose. They note that the more explicit the cues are, the more readily they are encoded and the more effective they are in content retrieval.

Written texts also come in a wide variety of formats and styles, from newspaper articles, to books and magazines, to advertisements, and so on. These styles can concern not simply the text itself, but also the spatial layout in relation to the page it is on and the other pieces of text around it. That is, text has a physical and spatial presence as well as an abstract meaning, and needs to be made and used, as well as composed and interpreted (Waller, 1982).

Waller (1982) continues to note that text is a static and reflective medium where the argument presented can be carefully controlled by the writer, whilst its reception is controlled by the reader, who can turn over pages, re–read a section, or close the book. He also suggests that, while much typographic design is concerned with fashion and style, it should actually be the result of:

- the syntactic structure of the content, affected by the ways the items are ordered and grouped on the page;
- artefactual effects (e.g. page breaks) which, although having no semantic import, constrain various options available; and
- the way the text is to be used, e.g. as a tutorial guide or a reference work.

Therefore, the use of typography should add to the semantics of the text, either as an aid to understanding or as an aid to access. This use of typography is expanded on in the next section.

3.2.4 Further Uses of Typography in Text

Typography has been found to be of use in explicitly signalling the structure of a text. It also may serve other purposes. Waller (1982) has shown how typographical features can be used in relation to how text is accessed, how they may be used to add syntactic meaning, and how they can be used instead of punctuation.

3.2.4.1 Typography as an access structure

Contents pages, indices, glossaries and summaries are examples of logical structures in documents, used for *globally* accessing a text in different ways, which can benefit greatly from effective typography. Contents pages, for example, provide an overview of the structure of a book, and generally use typographical means to show the hierarchical structure of a document through spacing and weighting of the headings in a tabular format.

Typography may also be used for accessing text *locally* through headings and layout. Headings are generally used for orientation towards a block of text, allowing readers to find a piece and be reminded of its context while reading the prose under it. Text layout is also used for orientation, but more towards a single word, sentence, or phrase, such as through indented margins for a quotation or italicised words for emphases. Layout can also form structural orientation through the regular positioning of running headings and marginal notes, which subsequently form a familiar pattern from page to page.

3.2.4.2 Typography as macro-punctuation

Typographical techniques may be used in preference to some forms of grammatical punctuation. There are four main functions of punctuation which have parallels with typography and layout. These are:

- *Interpolation* is the insertion of a short component into a longer one, so that the continuity of the longer piece is not spoiled. For example, parentheses may be used in a sentence but, if the inserted component is too long, it may be placed into a footnote instead.

- *Delineation* is used to mark the beginning and end of a sentence, picture, block of text, or other structure. For example, a newspaper article could begin with a bold heading and end with a rule in order to show its beginning and end.

- *Serialisation* concerns the ordering of text components into clear sequences, which can be achieved through tabular methods.

- *Stylisation* refers to the indication of a mode of discourse differing from the main body of text. For example, a side issue may be put in a coloured box or use a different typeface.

3.2.4.3 Typography as syntactic structure

A writer also has the ability of using typographical effects as syntactic cues for giving a discourse direction and coherence. However, using typography to this end is a difficult task. Writers are not expected to be professional designers, and professional designers are not expected to understand and intervene in the content of the texts they handle.

3.2.5 Markup

Markup[1] is the term used to describe codes which are added to electronically prepared text to define the structure of the text or the format in which it is to appear (Bryan, 1989). All writing involves some form of markup. For example, spaces between words indicate word boundaries, commas indicate phrase boundaries, and periods indicate sentence boundaries (Coombs et al., 1987). This fact is often ignored, with markup generally regarded only as a requirement for using electronic word–processing systems, such as using a control code to indicate a change in font. Currently, with the advent of WYSIWYG word–processing systems, these markup codes are generally invisible to the user, with only their effect visible (such as with a change in font).

Coombs et al. (1987) note that there are a number of different types of markup which can be distinguished in an electronic–based document. These are: punctuational markup,

1 Hand–written changes traditionally made to a document for editorial or design reasons are known as "mark up", whereas changes made electronically are known as "markup". For this thesis, no distinction is made between hand–written or electronic methods, so the term "markup" is used for both cases.

presentational markup, procedural markup, descriptive markup, referential markup, and metamarkup.

- **Punctuational Markup**

  This concerns the marks which provide mainly syntactic information about written utterances (dashes, commas, periods, etc.). The use of punctuational markup is expected from authors, yet is fairly complicated and subject to numerous stylistic variations. There are also variations in the appearance of punctuation marks, and many marks can be regarded as highly ambiguous. For example, do you use spaces around a dash, or is a period representing an abbreviation or the end of a sentence?

- **Presentational Markup**

  This refers to markup used to indicate specific entities. For example, a paragraph can be distinguished through horizontal or vertical spacing, a list has either bullets or numbers at the start of each item, and chapters often begin on a new page (and may be explicitly labelled "Chapter"). Presentational markup, such as centring lines or highlighting phrases, are also often performed by the author in the body of a paragraph, and often it is this form of markup which causes most difficulties in the translation of a document for other uses (such as an article for two journals, each with different layout and formatting rules).

- **Procedural Markup**

  Procedural markup replaces presentational markup in many text–processing systems, and consists of commands indicating how the text should be formatted. Procedural markup is specific to a particular text formatter and style sheet, and is device dependent. Procedural markup is most common in electronic batch text formatters, such as troff and TEX.

- **Descriptive Markup**

  Descriptive markup consists of tags which identify and delimit an element type. Descriptive markup is often confused with procedural markup, especially when macros

are used by procedural markup languages. The primary difference, however, is that procedural markup indicates what a particular text formatter *should do*, while descriptive markup indicates what a text element *is*.

- **Referential Markup**

  This refers to markup which is replaced by external entities during processing. Examples are abbreviations which are then expanded on, device–dependent punctuation, or a picture or block of text stored in a separate file. Referential markup is often supported by text processing systems through the use of variables or an *include* or *embed* command.

- **Metamarkup**

  Metamarkup enables the possibility for extending the vocabulary of a descriptive markup language. This may be used to define default attributes for element tags, or valid values for them (a valid date format, for example). Metamarkup may also be used to create new element types, thus allowing for a descriptive markup language to be adapted to suit the need of the writing environment.

Thus, markup can be used to separate the document *structure* from the document *form* and, depending on the type of markup used, may need to be changed if a new output device is used (as when using presentational or procedural markup). Of the different types of markup detailed above, Bryan (1989) and van Herwijnen (1990) note that markup can be split into just two broad categories: *specific markup* and *generic markup*.

Specific markup refers to instructions used to describe the format of a document which are specific to the programs used to generate or output the text. Presentational and procedural markup fall into this category. *Generic markup*, as postulated by Goldfarb et al. (1970), refers to instructions which identify the purpose of a piece of text, rather than its appearance. This is comparable to the use of descriptive markup described above, and again shows the split between the structure and appearance of a document.

3.3 Writing

As mentioned previously, writing is a complex task, as is the retention of written texts. This section will first look at how a text can be designed for improved access and retention, followed by a cognitive model of writing for an individual, and concluding with computer–based tools which may be used as aids to the writing process. This section models the writer as writing as an *individual*, with aspects of writing in a group detailed in the following section.

3.3.1 The Design of Text for Access and Retention

There has been considerable research focusing on the content of prose in order to make it more manageable and understandable for the reader. Whilst a computer system cannot force a specific way of writing onto a particular author, it can aid in the writing process through the use of clearly defined structures for bodies of text as well as guidelines for how writing or discourse should be presented, depending on the readership.

For effective retention, one of the basic problems facing a reader is that of *focus*. That is, the problem of selecting important information from the text. Duchastel (1979) notes that a reader must not only focus on the important points in a text, structuring them in some meaningful way, but must also be simultaneously processing new inputs which may be of only secondary importance. Reder and Anderson (1980) concurred with this view, finding that a summary was more easily remembered than a full text with side issues.

Halliday (1980), for example, has found that well–constructed prose has a central theme or focus, which is further elaborated on through supporting ideas in a passage. Thus, a paragraph which begins with a clear statement of theme sets the context for the discourse and makes it easier for the reader to understand.

Clark (1977) also noted a distinction between "*given*" and "*new*" information in discourse. Given information is information which has been mentioned previously or which can be inferred, while other ideas in the text are new, being introduced without any prior mention. Halliday (1980) found that as more new information was presented in a sentence, the sentence

became more difficult for people to process. Thus, "Carol bought a historical romance novel yesterday" was found to be harder to understand and remember than "Carol bought a novel yesterday. It was a historical romance." This is because in the second statement the novel is given information ("It") and thus does not require any special processing.

As well as these types of general rule, Sari and Reigeluth (1982) note that it is important to realise that writing for different purposes requires different styles and formats. Thus newspaper articles have one style, essays have another, short stories have yet another, and so on. On this same theme, Hirsch (1977) emphasises the importance of tailoring writing to the requirements of a particular audience, stating that "a shrewd decision about the knowledge that the writer can tacitly assume in his audience may be the most important decision the writer makes".

Rhetorical functions – the writer's means of expression	
About the argument	Summarisation (title, summary) Introduction (forward, preface, introduction)
Within the argument	Emphasis (underlining, italics, etc.) Transition (headings, space) Bifurcation (lists, parallel texts)
Extra to the argument	Substantiation (footnotes, appendices, references) Addenda (acknowledgements)

Access functions – the reader's means of enquiry	
About the book	Overviews (contents list, abstract) Definitives (glossary, index) Identifiers (title, author, style)
Within the book	Locators (topical headings, signalling) Descriptors (functional headings, captions)
Extra to the book	Study guidance (recommended reading, exercises)

Table 1: The dual role of typography and typographically signalled text components.

Adapted from: R. Waller (1982) "Using Typography to Improve Access and Understanding."

Psychological research has also shown that pictures and illustrations are better remembered than words and, whilst some detail in a picture may be lost, enough is remembered so as to

reconstruct the meaning or sense of the picture (Paivio, 1975; Franken & Rowland, 1979). This thesis is not concerned with aiding writers in their design of illustrations. For a comprehensive set of design principles for illustrations see Winn and Holliday (1982).

Finally, as noted in the previous section, typography can be used as a means for a writer to express an argument, or as an access method for the reader. Structural elements of a document, along with typographical cues, may also be used as rhetorical functions for a writer and access functions for a reader. These functions are outlined in the table above.

3.3.2 Cognitive models of writing

As well as noting how writing can be improved through syntactic constructs, use of language, and typographical cues, we also need to look at the cognitive processes involved in the act of writing. Much of this research has been conducted by Linda Flower and John Hayes (Flower and Hayes, 1980; Flower and Hayes, 1981; Hayes and Flower, 1986) who, having studied writing as a problem–solving process, have developed a model of the mental representations and operations of writing.

This model identifies three main component processes: *planning*, *translation* and *reviewing*. Figure 9 below outlines these component processes, along with the main sub–processes they contain. The model is iterative, with new ideas leading to new goals and new phases of planning and translation. The writing process is also not a fixed sequence of steps – instead a writer may engage in either some or all of the activities shown below (Coke, 1982).

The planning process relates to the setting of goals and to generating and organising information relevant to the task. Goal setting may include topic choice, selection of writing style, or choice of target audience. These goals are generally established at the start of the writing task, setting a frame of reference for the writer and providing constraints for his or her decisions while writing. Gathering information concerns collecting the information required on the subject matter and making notes on it. This leads to the organisation of the material, either mentally or through an outline of the subject. Pattern notes, as detailed previously, may be used at this stage in helping to organise the information for use.

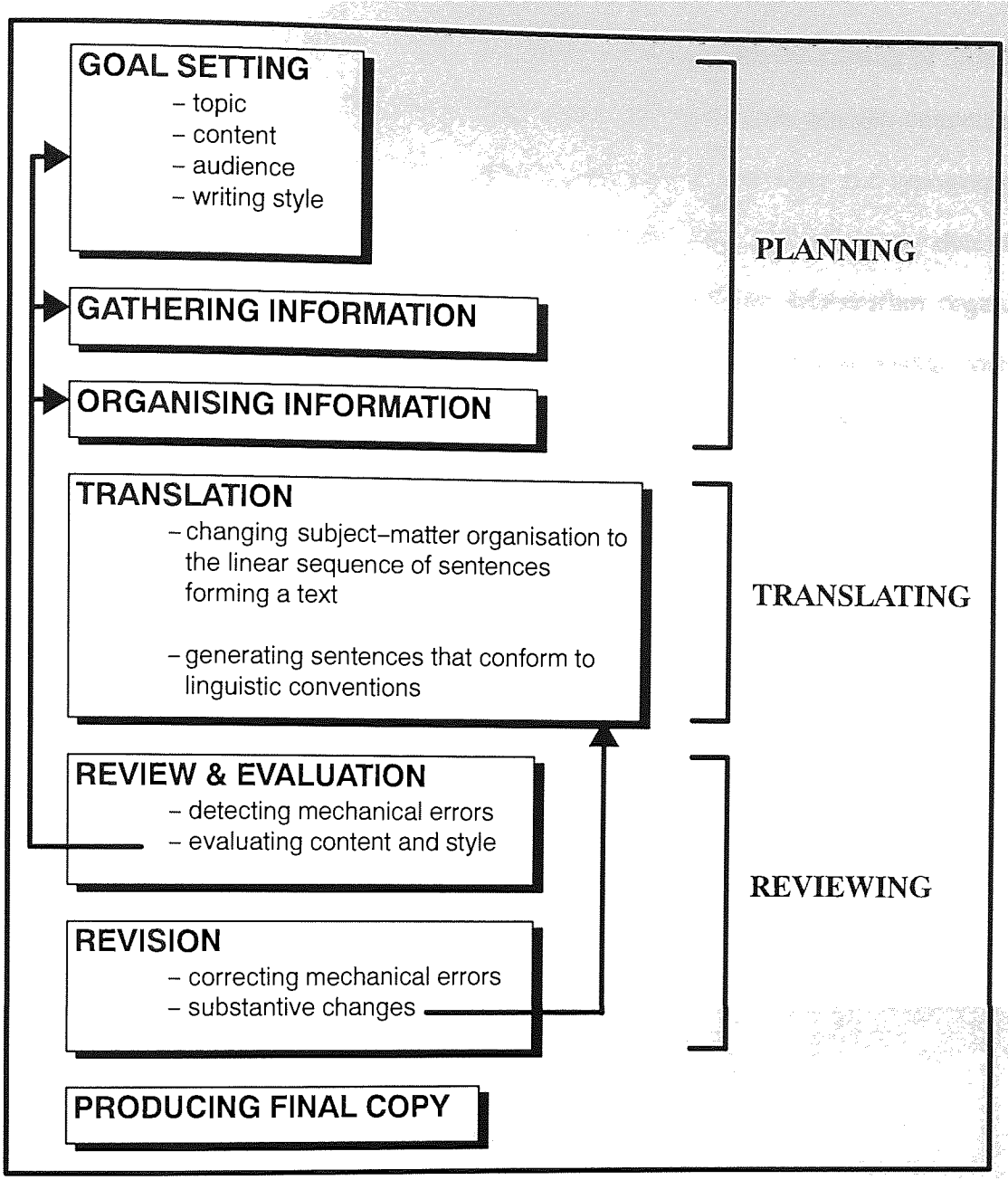

Figure 9: Components of the writing task with arrows suggesting optional return to other components from review and revision.

Adapted from: E. Coke (1982) "Computer Aids for Writing Text."
In D.H.Jonassen (Ed) *The Technology of Text* p385

Translation involves the conversion of plans and ideas into text in order to meet the goals. This involves the actual creation of sentences of text, deciding what to write, choosing how to structure the paragraphs, and formulating whether certain information should be presented in tabular or graphic form. The writing itself depends on the writer's choice of discourse style, and is carried out according to the linguistic conventions of the language being used (Coke, 1982).

Reviewing refers to the evaluation of the text produced along with the editing of either the text, or the ideas and goals relating to it. The review and evaluation process concerns the detection of mechanical errors (spelling mistakes, grammatical mistakes, and typographical errors) and the evaluation of a document's content and style. This evaluation may determine whether the subject matter has been covered adequately and the information organised effectively, or if it is in a style appropriate to the chosen audience. The revision stage corrects those mistakes identified, followed by making any substantive changes to the document. This leads back to the translation stage of the writing process till the document is satisfactory for publishing as a final copy.

Finally, Swigart (1990) notes that writing is not an entirely organic process, but is shaped by its tools in ways unconscious and unexpected. The movement of the hand manipulating the pen, the play of muscle and tendon as the mind directs words to flow from the tip etc., all impact on the outcome of the writing. How long is a chapter, for instance? That may depend on how fast the writer can write, how close his deadlines may be, or how much space he has for his article.

3.3.3 Computer support for writing

Computer systems may be used to help with the writing process. The past ten years have shown considerable advances in the development of word–processing systems for personal computers but, as it becomes easier to physically create documents, so there is a greater need to have sophisticated document management systems for handling these increasing volumes of text.

Computers can support writing in all levels of the writing process. They do not replace the writer, however, but assist him or her in performing well–defined repetitive tasks, or tasks requiring the retrieval and organisation of quantities of information that strain human capacity. Ambiguous tasks are left to the writer, resulting in the computer and human each doing those activities they do best (Martin, 1973).

3.3.3.1 Word Processing Systems

Microsoft Word ™, WordPerfect ™ and Ami Pro ™ are examples of computer packages which are widely used on personal computers by people involved in writing anything from a simple memo to a complex technical document. These word processing systems each provide a wide range of advanced features to aid in document production. For example, an outliner can be used in the ideas generation and planning stages of a writing project. Section, table and bullet generators, along with complex editing facilities, (such as automatic style sheets and 'drag–and–drop' editing), can also aid in the writing and layout of the document. Finally, spelling and grammatical checkers are aids for proof–reading and reviewing the document, with scaleable WYSIWYG ("What You See Is What You Get") displays useful for formatting and previewing the final output. These systems may all help reduce the "cognitive strain" (Flower & Hayes, 1980) placed on the writer while producing text.

A phenomenon which has been found to occur when using advanced WYSIWYG word processors however, especially in the learning phase of a system, is that of *fontitis* (van Herwijnen, 1990). Here, people use as many fonts as possible in designing and laying out a page, with elements representing similar functions being typeset in different fonts. This 'creativity' in layout tends to result in an incoherent collection of poor looking documents.

3.3.3.2 Batch Text Processing Systems

An alternative to these word–processing systems is the use of computer tools to support specific writing tasks. TEX (Knuth, 1984) and LaTEX (Lamport, 1986), for example, is a mathematical typesetting language developed to create well–formed formulae using a special notation for typesetting mathematical expressions. Thus,

```
$$y = {{ \alpha Q_c^2 } \over 24 } \left| A.$$
```

in TEX produces:

$$y = \frac{\alpha Q_c^{\,2}}{24}\left|A.\right.$$

Software is also available for batch text processing so that, starting from a text file, a system can first scan the text for spelling mistakes, then format the text for whatever purpose it is required, and then output the text to a printer.

3.3.3.3 Markup Systems and SGML

As mentioned above, a solution to concerns about the formatting of a document is through the use of markup, especially generic markup. Word–processing systems such as Microsoft Word™ (Microsoft Press, 1995) adopt one approach to this problem through the use of *style sheets*. Here, sets of formatting can be grouped under user–defined names, such as 'Heading1' or 'ComputerQuote', and then applied to appropriate paragraphs. The style sheet metaphor, however, still orients authors towards the *presentation* rather than the *role* of entities in a document (Coombs et al., 1987).

```
<!-- Document type definition for a memo                          -->
<!ENTITY % doctype "memo"                                         >

<!--  ELEMENTS   MIN CONTENT (EXCEPTIONS)                         >
<!ELEMENT memo   - - ((to & from), body, close?)                  >
<!ELEMENT to     - O (#PCDATA)+                                   >
<!ELEMENT from   - O (#PCDATA)                                    >
<!ELEMENT body   - O (para)*                                      >
<!ELEMENT para   - O (#PCDATA)                                    >
<!ELEMENT close  - O (#PCDATA)                                    >

<!--   ELEMENTS NAME          VALUE              DEFAULT          -->
<!ATTLIST memo status  (confidential|public) public             >
<!ATTLIST from   who           NAME           Brian              >
```

```
<!DOCTYPE memo PUBLIC "-//Quorum//DTD
Memo//EN">
<memo>
<from>Brian
<to>Martin
<date>5th June 1995
<subject>Board Meeting
<para>Unfortunately I have had to reschedule the board
meeting for 10.00 on Tuesday. I hope this does not
inconvenience you too much.
</memo>
```

Figure 10: A document type definition for a memo, and an example of its use.

The *Standard Generalised Markup Language* (SGML) is a markup language which has been designed specifically for the creation of diverse types of documents. Developed primarily by Charles Goldfarb at IBM (Goldfarb et al., 1970; Goldfarb, 1990), SGML has been adopted as international standard ISO 8879 (International Standards Organisation, 1986) for text information processing and document interchange. SGML separates the structure and layout

of a document by using descriptive markup to identify elements in a document. It uses a *document type definition* (DTD) to determine what elements are available, whether they should have default attributes, and in what order they should occur. Figure 10 above shows a typical document type definition for a memo, along with an example of its use.

The advantage of using such a DTD is that the integrity of the document may be maintained. For example, if specific fields need to be completed, (such as the 'to' field in the memo), then the SGML parser will cause an error if the field is left empty.

SGML also allows for sub–documents inside a document, along with the existence of different forms of the document, to aid the document's integrity and reusability. It also embodies both the text and structure of a document in a single text file, which can be easily transferred, distributed, and reproduced between authors.

3.4 Group Writing

Collaborative writing is a common occurrence in industry, where large groups of documentation and paper form the backbone of a company's information infrastructure. In academia collaboration is also widespread, with books and journal articles written by research groups or co–authors. Fewer examples can be seen in general literature, however, where our culture values individual responsibility for ideas and promotes the ideal of the lone author struggling for self–expression (Sharples, 1993).

This section will examine writing amongst a group of people. It will begin by considering some of the practices which have been adopted by writers, followed by models of writing by groups. To conclude, it will describe some computer systems which have been developed to help support the writing process amongst groups.

3.4.1 Collaborative writing practices

In a collaborative writing project, authors need to offer ideas and be willing to accept the consensus of the group. In this matter, group dynamics and social practices form a major part of the work – if individuals feel unable to trust one another, the "collaborative" aspect of writing may be lost, resulting in simply a collection of individual works. Sharples (1993)

notes, however, that if the collaborative ideal can be grasped and the authors can work to overcome conflict, coordinate their activities, and arrive at a shared understanding, a text may be created which is more than could be created by any single person.

"Collaborative writing" can be regarded differently by different people in the extent of their *togetherness* in the project. Some regard the simultaneous writing of a document to be collaborative, while others regard getting comments and annotations on a piece to be collaborative. Rimmershaw (1992) identified three ways of collaborative writing in a study she undertook. These were:

- **Writing Together**

  Here, people physically work together on a document, be it on computer or with a pen and paper. These are 'intense' writing sessions, sometimes reserved by writers for key stages in the writing project.

- **Exchanging Drafts**

  This is a common practice in collaborative writing, where collaborators exchange ideas or blocks of text at specified intervals. This allows each person to apportion the work to their own needs in their own time. Often, one person will be assigned the role of writing the first draft, which is subsequently commented on and added to by the other collaborators. These drafts or normally exchanged by either post or electronic mail, with annotations then marked up on the draft, or exchanged via electronic mail, facsimile, or some other technology.

- **Meeting Needs and Circumstances**

  Finally, the form of collaboration chosen is due to the current needs and circumstances of the collaborators. For example, a group may each write a separate piece on a similar subject which is 'blended' 'connected' and 'rearranged' at a later meeting. Depending on the success met, this method may be used again by the group members. If time is a constraint, however, this method is unlikely to be adopted – instead a delegated style of 'filling in the blank sections' is more likely to be used.

In these three main practices technology plays a minor role, with writers adapting their practices depending on what technology is available. For example, if a group is using the same word–processor and is working by exchanging drafts, then the drafts may be exchanged electronically on disk instead of on paper. Thus, the technology available may constrain a set of writers to adopt a particular practice. Rimmershaw (1992) concludes that, as the social practices are prime and the technology secondary, that any tool or system to support a writing group should allow that group to do what they *actually* want to do, and not force them into unwanted modes of practice.

3.4.2 Models of collaborative writing

Collaborative writing can exist in many different forms, from taking a few minutes to create a joint memo, to a number of years for writing a co–authored book. Ideas gained while *not* writing, such as talking with colleagues over lunch, may also be incorporated in a document. Thus, any model for collaborative writing needs to take this range of diversity into account.

Sharples et al. (1993) have identified three types of coordination which occurs in collaborative writing, and which can be used as the basis for a collaborative writing model. These coordination strategies for writing are *sequential*, *reciprocal* and *parallel*, and may each be adopted at any stage of the writing process. Figure 11 above shows these coordination strategies graphically.

Sequential working occurs when the collaborators divide up the task among one another, with the output from one stage passed on to the next writer in line. Thus, writers can write separate sections, or individually review completed drafts. This form of collaboration generally takes place under a single editor, and is common in magazines, newspapers, and edited academic books. Here, the authors take responsibility for their individual work, having negligible input into the work of others, with the editor being responsible for maintaining a common style and controlling the volume of contributions. Comments and advice may be freely given on each others' work, however, which gives the final output a sense of consistency.

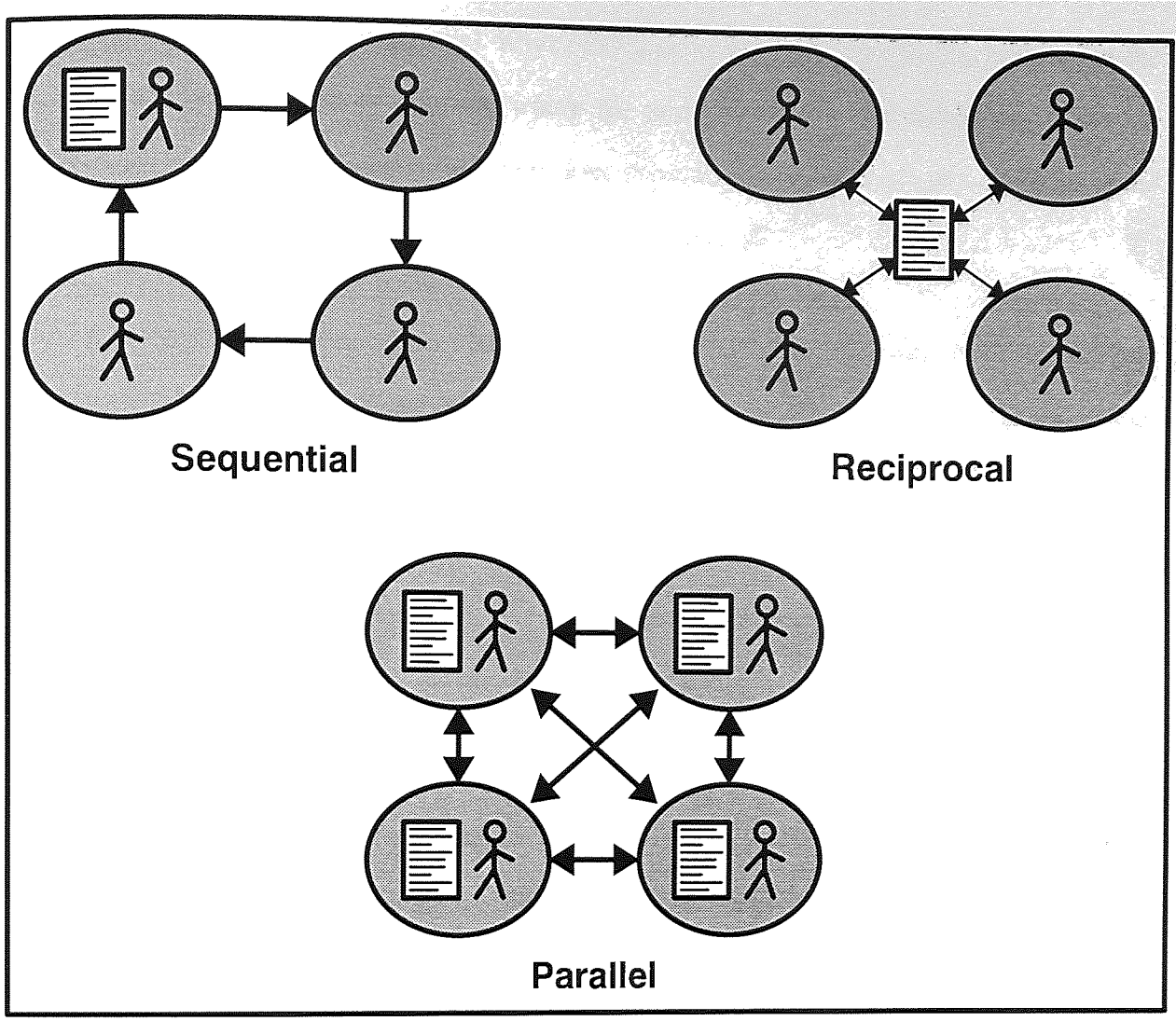

Figure 11: Strategies for coordinating collaborative writing.

Adapted from: M. Sharples, J.S. Goodlet, E.E. Beck, C.C. Wood, S.M. Easterbrook and L. Plowman (1993) "Research Issues in the Study of Computer Supported Collaborative Writing." In M. Sharples (Ed.) *Computer Supported Collaborative Writing* p.15

Reciprocal working has the collaborators working together to create a common document, adjusting their activities to take account of each other's inputs to the discussion. Brainstorming (Osborn, 1957) is an example of this, the focus of the session often being a whiteboard or flipchart. Drafts of text may also be produced using this method, either by one member typing and the others 'thinking aloud', or through access to a common computer with a central file. Rimmershaw (1992) has noted how this method may be used, as in the practice of *writing together.*

Parallel working divides the task into a number of sub–tasks, with the collaborators working in parallel to accomplish the tasks. Assigning separate sections of a document, or putting a document through a separate writing process (such as spell–checking or reviewing), allows

the collaborators to work in parallel. Kaye (1992) notes how this approach is adopted in the design of Open University courses, where the work consists of an initial stage of goal definition, followed by an execution stage done in parallel by individuals or small teams, and concluding with an integration stage of the work produced.

All of these modes of writing may be iterative. For example, with parallel working, lawyers drafting a contract or treaty may separately 'build up' the contract, with each integration stage consisting of an agreement between both parties of what is included up to that point. The process then iterates until an agreement is reached for a final draft on the contract.

3.4.3 Computer support for collaborative writing

Along with the research on collaborative writing, there has been a large increase in the use of network technology in the workplace. Such a growth has added to the use of groupware tools, detailed in the previous chapter, to both aid and enhance group working. This section will briefly look at some of the computer systems which have been developed to support collaborative authoring.

3.4.3.1 NoteCards

NoteCards (Trigg et al., 1986; Halasz, 1988) is a hypertext system developed primarily for the manipulation of ideas, but has also been adapted for use in collaborative authoring. To this extent, it is only useful for small workgroups, having limited facilities for extensive collaborative writing.

The system is based around the 3x5 paper notecard, and provides links and browser facilities between the cards. It is also able to give a general overview of where each card 'fits in' with the document, and uses a shared workspace for collaboration. NoteCards has a limited version control system, and does not tackle the problem of work partitioning and allocation among the writers, and is best viewed as a basic annotation system.

3.4.3.2 Quilt

Quilt (Fish et al., 1988; Leland et al., 1988) is an asynchronous collaborative authoring system. It is used to manage the collaborative aspects of group authoring, and is primarily designed to deal with the issues of coordination and information sharing between the group members. As a result, the system assumes that people have different roles in the production of a document, and assigns different privileges to these roles. This facility provides a degree of awareness amongst the group, as they can see what actions each member is able to perform.

Quilt also contains a system and user log. The user log consists of messages left by writers, either detailing what they have done, or what they intend to do. The system log details exactly what a person has done on the system, consisting of messages such as:

```
"Edited section 2.2 for 2 hours; 15 to 45 paragraphs
changed"
```

The document structure of the system consists of a basic tree containing a base document with *n* annotations, which in turn contain further annotations, and so on. These annotations can be text or voice, and consist of either private or public comments, or a message directed at a specific author or group of authors.

3.4.3.3 Groupwriter

Groupwriter (Malcolm & Gaines, 1991) is an asynchronous collaborative authoring system based on the notion that any tool developed for collaborative writing should contain all the features of existing commercial word processing systems. Groupwriter is based around a commercial word processing engine and has full versioning capabilities, being capable of maintaining parallel and reproducible versions of the document being produced, and allowing different versions of the document to be compared with each other.

The user interface of the system has also been designed to hide the complexities of the version control system underlying it. This means that an author may use the system as a simple word processor, yet have the ability to switch to historical versions of the document if required. The

system, however, has no concept of role or the locking of work produced, which may lead to potential conflicts.

3.4.3.4 ShrEdit

ShrEdit (Olson & Olson, 1991), standing for "Shared Editor", is a synchronous group writing system which allows for the fine–grained editing of text, prescribing no specific structure for collaborative work, relying on the authors to form some method of working together. The system incorporates a shared (public) and individual private views of the document, and allows all writers to view and change the same public document, with all participants able to type simultaneously within one character position of one another, locking only occurring at the level of text selections.

ShrEdit also has the ability to lock views as a method of tight coupling between users, and also allows one user to 'watch' what another is doing at the time. This may result in one participant watching what the other is writing as a method of gaining ideas for their own area (Olson et al., 1992).

3.4.3.5 PREP

PREP (Neuwirth et al., 1990) is an asynchronous "work in preparation" editor, which can be used by groups to develop documents together. The system concentrates primarily on the early stages of the writing process, and presents its text as parallel columns denoting varied forms of the text. Thus, four columns could be used to show a document plan, a draft text, and two sets of annotations by two authors. This interface is analogous to that of the writing and commenting on of religious texts by monks in the middle ages, who used a columnar style for amendments to the original work.

PREP again uses roles as a means of determining author responsibility and controlling access, and has been extended to allow the granularity of the notification of changes to be controlled by the authors (Neuwirth et al., 1992).

3.4.3.6 Cooperative SEPIA

Cooperative SEPIA (Haake & Wilson, 1992) is based on the SEPIA hypertext system developed at GMD–IPSI. It consists of various activity spaces used for the structuring, planning, arguing, and writing of documents from a rhetorical perspective. The system is modelled on three modes of collaboration: *individual work*, *loosely coupled mode*, and *tightly coupled mode*.

Individual work occurs when an author is working on a node currently not being visited by other authors, loosely coupled mode is when multiple authors work on the same composite node, and tightly coupled mode occurs when the co–authors share the identical view of the node's contents. Cooperative SEPIA provides for a smooth transition between these different modes of collaboration, with automatic node locking and author awareness in both loosely coupled and tightly coupled modes.

3.5 Conclusion

This chapter has looked at the processes of document structuring and planning, writing, and text manipulation. It shows how research on document and writing structure, along with markup, can be integrated with typography to enhance the development of books and documents so that they are easier to browse, read, and comprehend.

The chapter also examined the difficulties inherent in the actual writing process, and the cognitive demands it places on a writer. It looked at methods of reducing these difficulties, and notes how computer tools for writers should have facilities to help ease this cognitive burden. Finally it considered writing amongst groups, recognising that it is a very social task, and concluded by briefly mentioning some collaborative authoring systems and the main problems of collaborative working they are focusing on.

This concludes the literature review. The following chapter will lead from this point to determine the main issues which a collaborative authoring tool should address, and present a model for collaborative writing.

Chapter 4

The Design of a Collaborative Authoring Tool

4.1 Introduction

Vertelney (1990) has noted that for effective collaboration to occur database, communication, and user–interface technologies must all be sewn together. With the introduction of collaboration technology, however, Grudin (1989) notes that there is a potential to both enhance and disrupt the psychological and social dimensions of work situations.

Thus, while careful design needs to be adopted in the design of any software tool, it is especially important for that of a collaborative tool. This chapter relates to the design of such a tool for collaborative writing, drawing on the research on group dynamics, communications, decision–making and negotiations presented in chapter two, and on the areas of writing, typography, and document structure presented in chapter three.

This chapter will first consider the issues which need to be addressed with a collaborative writing tool. Next, it will present a model for collaborative writing, concluding with a description of the flow of communications between writers in a collaborative endeavour.

4.2 Issues relating to Collaborative Writing

As has been previously discussed in the literature on groups and writing, there are many different ways for a group to work together, along with a wide variety of methods for both individuals and groups to progress with a writing project. Therefore there are a wide range of issues which any tool needs to address if it is to aid the collaborative writing process between authors.

There are many different and varied forms of writing task. Subsequently, issues important in one case may be less important in another. Thus, changing code in a multi–user programming environment, making notes in group meetings, negotiating and drawing up a contract, and

jointly writing an article or book are all examples of collaborative authoring scenarios. Therefore, just as there are different groupware systems to aid in different ways of working, so there needs to be different collaborative authoring tools for different authoring scenarios.

This section looks at some of the major issues which need to be addressed by a collaborative authoring tool. Again, all issues may not need to be addressed in all authoring scenarios and may, indeed, hinder the working practices of the collaborators. Any rules and "ways of working", such as access controls to others' work, should thus be able to be enforced or relaxed as required by the authors.

4.2.1 Cognitive Models of Writing

As identified in chapter three, research has shown *how* people write, and the processes which they go through to achieve a written text (Flower and Hayes, 1980; Rimmershaw, 1992; Sharples et al., 1993). From this research it has been found how people 'jump' from one cognitive process to another, such as writing in one instance, to generating new ideas in the next, to reviewing what has already been written.

Writing is also an open–ended and under–constrained task, requiring the juggling of numerous constraints. Work on pattern notes (Fields, 1982) has shown just some of the constraints (deciding the presentational viewpoint, naming the key issues and ordering the key issues) which a writer needs to take into consideration when undertaking a writing project. Many other constraints are also evident, such as work deadlines or determining the required length of the prose, which need also be considered throughout the endeavour. Sharples and Pemberton (1990) have also noted how, for a writing task, there is neither a fixed goal nor a formal transition between states, but rather there are many possible texts which could fit a writer's goals and numerous possible actions which a writer could take at any stage.

Thus a collaborative writing system needs to support the writer so that it can harmoniously switch between different modes of use, (such as brainstorming in one instance to reviewing in the next), without any additional effort on the part of the author, and independently of each author.

4.2.2 Group Awareness and Conflict Resolution

During the life of a collaborative document, and by the actual nature of it, conflicts may arise during its production. These conflicts may occur for a number of reasons, such as differences in opinion over content, social problems in the group, overlapping roles and responsibilities, and replicated work.

Most collaborative systems take a vague view regarding the areas of conflict and conflict resolution, relying primarily on:

- **Group Protocols**

  Systems such as ShrEdit (Olson et al., 1992) rely primarily on formal or informal group protocols for collaborative working. The system enforces no specific access rights on any part of the document, allowing any author to add, view, and edit any portion of the document. This 'openness' is a deliberate design consideration in order to allow maximum freedom between collaborators, and has been found to work well in design meetings where the collaborators are working synchronously in close proximity for a short period of time. The system is not designed, however, for the creation and integration of large documents with life–cycles of potentially many years.

- **Authorship Access Rights**

  Quilt (Leland et al., 1988) is an example of a system which uses enforced access privileges to manage the authorship of work and, like similar systems, matches author roles to access rights. For example, an 'author' will have read and write access to their text, while a 'reviewer' will only have read and annotate access to the text. The researchers have commented, however, that the system of access rights needs to be more accessible and flexible, allowing for the rapid switching of the role which an author may take on (such as editor, then writer, then reviewer).

- **Automatic creation of new Document Versions**

  A final option adopted by many systems is simply to create a new version of a document whenever it has been changed. Groupwriter (Malcolm & Gaines, 1991) is one such system

which creates new document versions as it is changed, allowing for versions to be 'rolled back' if required. The system does not help in the process of rebinding the different versions into one draft, maybe keeping the latest version of one paragraph but rolling back another. Instead, this task is left to the collaborators, who need to decide which versions to keep, which in turn assumes that all the co–authors are working harmoniously together in the project.

Collaboration is not a purely harmonious endeavour, however, with conflict playing as important a role as cooperation. It is often the history of these conflicts which shapes a document, so computer systems to support authors writing together should assist in negotiation and conflict–resolution matters as well as the basic writing aspects of the project.

Some conflicts occur simply as a result of poor coordination between group members. This can manifest itself in a lack of awareness among members so that they do not know what one another is doing, potentially leading to one writer replicating the work of another, or two authors writing on the same theme in parallel. Dourish and Bellotti (1992) have noted that, in a collaborative endeavour, authors need to know what each are working on, not only in *content* but also in *character.*

Group awareness may be achieved by a number of means, from shared views, telepointers, and audio or visual means in a synchronous system, to electronic mail, history logs, and change bars in an asynchronous system. These forms of awareness can help reduce writing conflicts, so that authors know what each person is working on, or on what they intend to work.

As mentioned above, other forms of conflict may occur in groups, such as conflict due to differences of opinion on how to proceed with a topic. Collaborative systems tend to simply ignore these problems, leaving the resolution of conflict to social group pressures. Instead, the computer should support conflict resolution techniques using, for example, those methods of controlling group structuring and negotiations (nominal group technique, delphi technique, voting practices) as described in chapter two. Negotiations should also be recorded as an additional source for extracting the history of the document's production.

4.2.3 Document Form

Documents can take on many different forms. Depending on the nature of the document, it can be viewed as a series of linked nodes (as in hypertext systems) or as a single page–based document (as seen in most word processors).

Each viewing method has its advantages and disadvantages, with specific applications generally lending themselves to one method or the other. By using the hypertext form, browsing can be very disjoint as various links get traversed in order to find the body of the text. Hypertext systems, as mentioned by Halasz (1988), are generally tailored to support either authoring or browsing, not both. Graphs representing document nodes can also quickly become large and unwieldy as document sizes increase, although Feiner (1988) notes that graph layout programs using graph trees, indentation, or nested set notation, as well as the suppression of sub–tree details, can improve the situation.

Hypertext views can be advantageous if the document is disjoint, requiring the reader to move to different parts of the text, or if the content is suited to being distinctly separated into small units. For example, technical texts may have direct links to glossaries of technical terms, or could link to a series of multimedia nodes depicting how something happens or how to achieve some task (such as a video clip showing the results of a chemical experiment, or how a piston engine works). This form of multimedia authoring, which is increasingly more involved with the use of cinematic methodologies is, however, beyond the scope of this thesis.

Page–based document views are more conventional and familiar for readers, and are easier to read than a hypertext system if the document is read in a linear manner. Browsing, however, is made more difficult, unless there is an extensive table–of–contents section, or if all that is required is a quick scan through all of the basic text (epitomised by '*flicking through the pages of a book*').

Regarding access rights to the document, a document whose form consists of hypertext node–link structures can use a node as a basic unit for which authors can be given specific rights. Such nodes are also easy to identify in cases of author overlap, with it being trivial to determine which authors are working in what nodes. With page–based views, however, the

assignment of authorship rights to specific 'blocks of text', which are yet to be written, is much more abstract (you cannot have empty nodes as in the hypertext case, for example). Overlapping rights between authors is also much more likely, and it is hard to determine what rights specific authors have.

A page–based document can, of course, have a node–link structure underlying it. Thus, a node–link view could be used for the assignment and display of author rights, and a page–based view generated automatically from this for reading purposes.

4.2.4 Document Overviews

Hansen and Haas (1988) have noted that writers need a *sense* of the text. That is, a:

> *"feeling a user may have that he or she has a good grasp of the structural and semantic arrangement of the text – the absolute and relative location of each topic and the amount of space devoted to each."*

Rimmershaw (1992) has also noted how authors find it important to be able to view a whole document at once, having a global view of it whilst being able to get quick access to the detail of the text itself at any point. This can manifest itself in many types of medium, not just through using a computer system, such as when Rimmershaw talks about a lecturer who had a collaborative meeting where "*our chapters were all over the floor*" – a means of seeing the whole document at once, and then selectively focusing on specific parts. Having a global view is invaluable in helping a reader find parts of the text, following the thread of an argument, or getting the gist of the material.

Severinson Eklundh (1992) says that a document overview can be either logical or physical, and active or passive. A *logical overview* is a way of emphasising the conceptual structure of the text to the reader or writer, either explicitly through the use of markup, or indirectly through the use of typography. Outliners, pattern notes, and hypertext graph trees can each be used to obtain a logical overview of the text. A *physical overview*, alternatively, relates to the appearance of the text through its typographical layout and spatial placement on a sequence of pages.

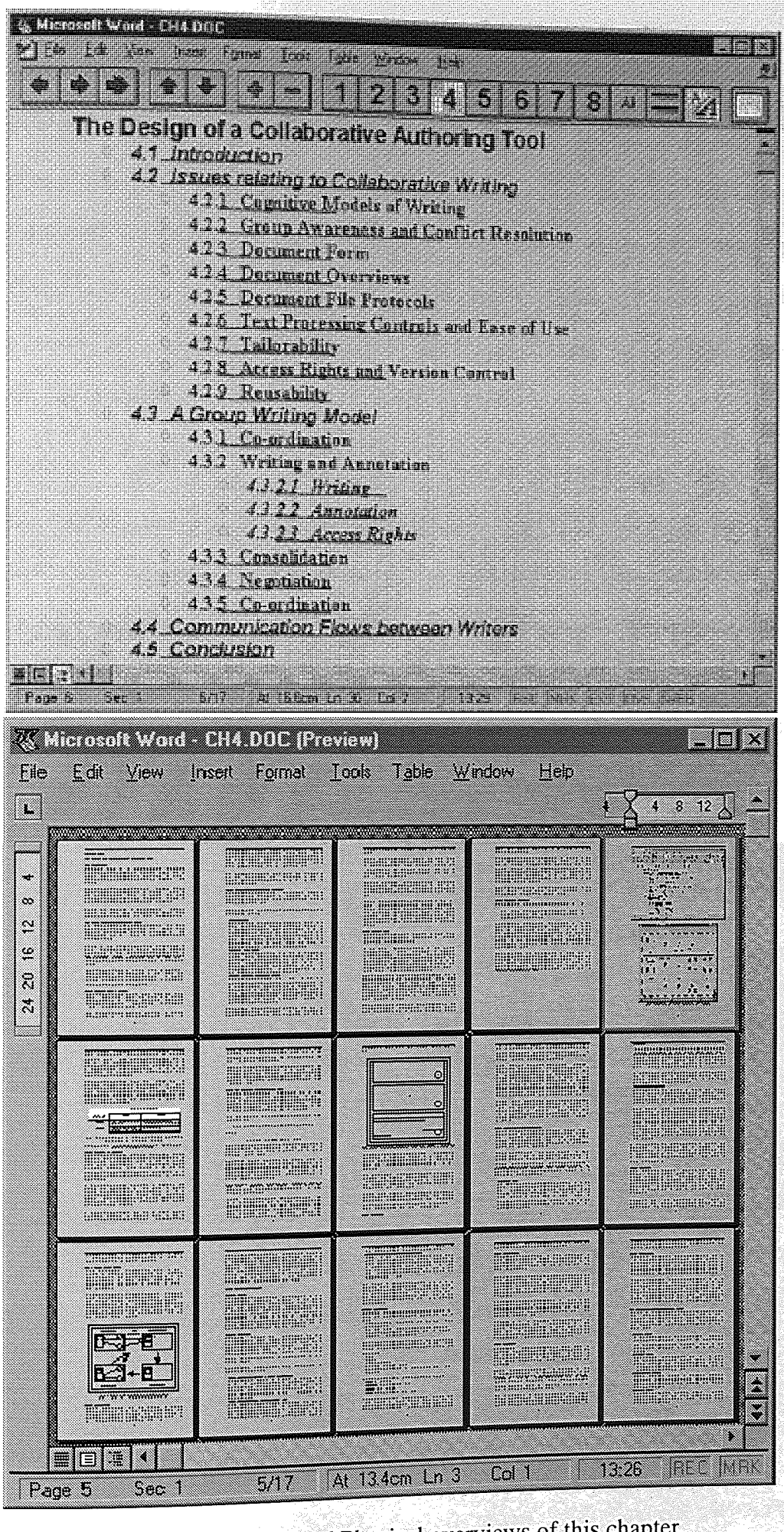

Figure 12: Logical and Physical overviews of this chapter.

Hansen and Haas (1988) have previously noted how word processors decrease the physical overview of text compared to paper, with the screen providing only a small window onto the text, textual positioning on the page not being constant, and repeated scrolling hampering the spatial sense of the text. As word processors become more advanced, however, some of these problems are being reduced. Outliners can now give a logical overview of the text, and page previewing tools can give an idea of the spatial layout of one or more pages through thumb–nail views of the pages. Figure 12 above shows an example of this, with both outline and thumb–nail views of this chapter.

As well as an overview being logical or physical, it may also be active or passive. An *active overview* is an overview which can be manipulated by the writer. Thus, an outliner which allows the re–ordering, promotion, and demotion of sections is an active overview. A *passive overview*, on the other hand, cannot be directly manipulated, and is used more as an external reference for the reader or writer, such as a table of contents page. Table 2 below shows examples of active and passive overviews, and the level of mental representation they afford to the writer.

	Level of Representation	
Type of Overview	***Low***	***High***
Active	Text written directly on paper pages	Idea processor with direct manipulation of 'collapsed view'
Passive	Paper print–out of text written in word processor	Table of contents

Table 2: Different types of overview of a written document, categorized according to level of representation and the active/passive dimension.

Adapted from: K. Severinson Eklundh (1992) "Problems in Achieving a Global Perspective of the Text in Computer–based Writing." *Instructional Science* 21, p. 81.

4.2.5 Document File Protocols

Word processors and text entry systems rely primarily on proprietary formats for storing their document content on disk. The flat structure of a file does not normally map onto the

hierarchical structure of a document, nor is it compatible with other word processing systems. While some common formats, such as RTF (Rich Text Format), have been introduced to ease the problem, they are very simplistic in that they contain little information about the structure and styles used to build up a document. Instead they are concerned primarily with having a document *look the same* between one word processing system and another.

For a collaborative authoring system, it is vital that authors can exchange work easily between one another. Authors prefer to use whatever tool they are familiar with, resulting in a plethora of different word–processor file formats. Attempts have subsequently been made to improve file formats for documents. PostScript™ (Adobe Systems Inc., 1990) is a language for interpreting how a page is displayed, either to the screen or other output device such as a printer. This, however, again relates to the *appearance* of a document. PDF, the *Portable Document Format* (Adobe Systems Inc., 1993), addresses this issue partially, including further semantic content along with the PostScript page descriptions.

As mentioned in the previous chapter on markup, SGML, the *Standard Generalised Markup Language* (Goldfarb, 1990), is a language which addresses the problem of document structure, leaving the appearance of the document to each individual writer's preference. Other markup languages have arisen which are primarily derived from SGML. HTML (Berners–Lee and Connolly, 1993), the *Hypertext Markup Language*, is currently one of the most widely adopted public domain markup languages, being the language of choice for the creation of World Wide Web pages on the Internet.

These markup languages, however, are common in that they are concerned more with the structure of the document than of the formatting, and that this structure is explicitly encoded into the document when it is stored on disk. This allows for easier interchange of the document between authors, with the document integrity maintained across the file transfer, the only difference being the formatting of the text, preferences for which are left to the individual writers.

4.2.6 Text Processing Controls and Ease of Use

Malcolm and Gaines (1991) and Sharples et al. (1993) have already stated that authors require advanced text processing controls in collaborative tools similar to those found in traditional word processors. Typographic, formatting, and document processing functions should each be provided by the tool to aid authors in the development of a document. Hypertext systems have a tendency to lack these facilities, instead having plain text entry or, at most, using simple formatting codes such as bold, italic, and underline.

Markup systems can get around the lack of expressiveness in a plain text entry system, using examples like:

```
He said, <q>A word's <b>bold</b> and one's <i>italic</i></q>
```

to produce:

He said, "A word's **bold** and one's *italic*"

This method, however, is not very satisfactory, requiring a writer to not only learn a new set of command tags for various functions (albeit with fairly obvious mappings), but also loses the visual feedback of current WYSIWYG word processing systems and is more cumbersome to enter at the keyboard.

Grudin (1989) has also noted that groupware tools often fail if they require members of the group to do additional work, especially if the members do not perceive any direct benefit. If a collaborative tool is more difficult to use than its single–user counterpart, or does not have features deemed valuable to the writer, then it will not be used in preference to the old system. Thus, tools like a spelling checker, thesaurus, and grammatical checker should be available for a collaborative authoring system.

As well as enhancing the expressive repertoire available to the writer, text formatting features can also be used as an aid to author awareness. Graphical formatting, for example, can be used as a visual cue to identify different users' writing in a text, through either colour, font, or style choice.

Regarding the use of colour for author awareness, one writer may choose to write in blue 'ink' while another selects black, red, or green. This enables the collaborators to easily see on the screen what each has contributed, but does not map well on paper or mono screens for external viewing. Inappropriate choices of colour can also cause difficulty in reading the text as, for example, with yellow text on a white background. Colours are also perceived differently by the eye, with rich colours appearing more prominent than pastel shades. Thus, colour choice can influence what the reader notices first when reading the text.

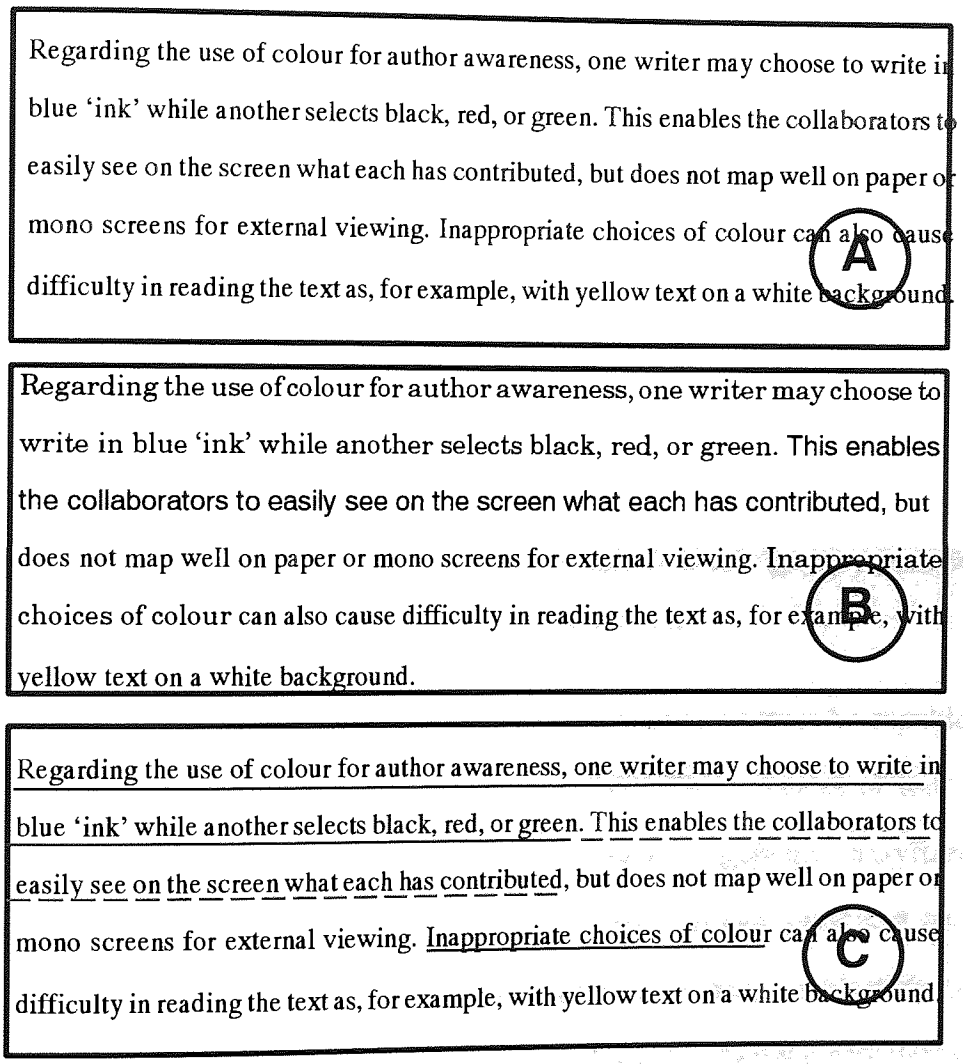

Figure 13: Colour, font, and underlining options for three writers working on a text.

Using different fonts has similar problems to that of using colour to differentiate authors' work, in that certain fonts will inadvertently direct the eye more than others, with the reading of the text again made more difficult. Certain emphases in the text can also be lost due to font changes, so that an emboldened word may get 'lost' if beside a font with a heavy weight.

Another method of determining what each author has written in a block of text is through a visual cue which does not actually affect the text content itself. For example, underlining of the text could be used, with different underline colours and styles used to represent each author. Depending on the granularity of the changes, other structures such as change bars using different colours or styles, could also be used.

It is, however, only with a rich set of text processing controls that any of these methods for improving author awareness can be considered. Figure 13 above shows an example consisting of a piece of text marked up using colour (example A), font changes (example B), and underlining in both different colours and styles (example C). From this figure it is easy to see the problems with each of these methods, as described above.

4.2.7 Tailorability

A group consists of a collection of individuals, each with their own views, preferences, and styles. Writing, in particular, can be a very individual task, with people having very set views and preferences on how they should do their writing, and how it should look as they are developing it. Forcing a collection of individuals to follow some 'group consensus' in the *ways* they work can often cause a collaborative system to fail.

Because of this individuality in a group, a collaborative system needs to be capable of being customised to the individual preferences of each person. Thus, choices of font style and attributes should be individually selectable, with each user having their own stylized view on the document text, yet still enable synchronous working between authors if need be. As mentioned in the previous chapter, using markup in the document can enable different authors to have their document formatting done differently, yet logically the document items mean the same. So, when the work is merged between the individuals, standardised group formats can be introduced across the whole spectrum of the document based on specified layouts or typographic design considerations as mentioned by Waller (1982), and not on an individual author's subjective opinions of what 'looks' good.

As mentioned previously, in addition to the customisation of the body of the text and the screen layout, the system should also be able to be customised to allow for alternate ways of

working between the group members. Therefore, if a group is very informal and participatory, no restrictions might be decided on for any of the writers, leading the system to being a very 'open' system. Facilities should, however, also be present that allow for the security and locking of text, both for reading and writing, if required by the group.

4.2.8 Access Rights and Version Control

No part of a document should be deleted. The history of a piece of text can form an important guide in any negotiations or conflicts which have arisen during its development. Thus, if one author overwrites another's work, a new version of the document should automatically be created which can then be *rolled back* at a later date if necessary. A document, therefore, can be regarded as consisting of a hierarchical tree with multiple revisions to disjoint portions of the document text.

The system should have the capability of limiting access to the document, so that writers can only make changes to portions of the text depending on what privileges they have been given. This access control is a major issue in collaborative work, and can be dealt with either through formal roles assigned to authors, or informally through social protocols. If formal authorship roles are used, however, it should be noted that they are often fluid, changing over the duration of the collaborative work. Thus the methods used for controlling access rights need to be easily changeable and adaptable. These rights can be controlled through various techniques, such as:

- **Inheritance rules**

  Rights can be set for either a specific portion of text, or can be inherited down through a complete section of the document. Logical hierarchical document structures aid in the setting of inherited access rights. Thus a writer could be given authorship rights to 'Chapter 3', which means that he/she has access to, and can create, any number of sections, sub–sections, and paragraphs within the boundary of 'Chapter 3'. Alternatively the writer could be given access to a specific part of the chapter, such as 'Chapter 3,

Section 4, Sub-section 1', which would only allow the writer to work on a very limited part of the chapter.

- **Delegation**

Access right delegation refers to a right which gives an individual the *ability* of granting and removing access rights for other group members. These rights cannot be propagated up a document hierarchy, however, so a person with delegated access rights for 'Chapter 3, Section 1' cannot give rights concerning 'Chapter 2' to another author. Thus, a document can effectively be divided into a number of smaller units, with managerial coordination decisions delegated throughout a project team resulting in a hierarchy of managers apportioning work to their subordinates.

If a collaborative writing project involves a large group of people, then the delegation of access rights becomes important. As in industry, work is often apportioned to smaller autonomous groups of a 'manageable' size which thus require the group leader to control the access privileges of those working for them.

- **Weak and strong rights**

Access rights can be applied either to specific individuals or to groups as a whole. A weak access right grants (or revokes) some form of access to a group of authors (or globally to everyone), while a strong access right grants (or revokes) some form of access to a specific person.

- **Positive and negative rights**

An alternative to *positive rights*, where users are explicitly *granted* rights and privileges, is *negative rights*, developed by Rabitti et al. (1991) for database systems, where a user can instead be explicitly *denied* rights and privileges to a portion of the document. So, an author could be granted a weak positive read right over an entire text, along with a strong negative right over a sensitive or confidential section.

Shen and Dewan (1992) further note that positive and negative access rights can ease the specification task of collaborative applications supporting dynamic roles. This is

especially pertinent as the life cycle of a collaborative document can span either a few minutes or many years. In the latter case, people may join and leave the project during its lifespan. If an author then wished for everybody to be able to read a section, except for a specific person, the section could be given a global positive read access right, and the aforementioned person a negative read access right for the text. Without negative rights, each person (bar the individual) would have to be explicitly given a positive access right to the section, as would each new writer upon joining the project.

All three access rights are useful for collaborative writing, as they can make a system as 'open' or 'closed' as required by the authors. Thus the system can be adapted to the working practices of the authors as a whole. Through delegated rights, global rules can also either be strengthened or relaxed, so that each sub-group working in a document can form their own individual norms for working, whether by means of informal cooperative means or formal roles.

4.2.9 Reusability

Parts of a document created should be able to be reused. The same body of material could, for example, be used as part of a textbook, a scientific paper, a memo, or in some other format, with few changes (if any) required by the authors. Electronic documents could also be generated as a by-product of the publication process for inclusion on electronic media. Work in the Space Standards Division at the European Space Agency (ESA) is currently progressing on this point, with a view of having international space standards available for public access on the World Wide Web (Kriedte, W. (1995), personal communication).

Having documents reusable in this way avoids unnecessary repetition of work by the authors. By having a central depository for the work, from which all revisions are made, document integrity is also improved as numerous unconnected instances of the document are not created and so errors are less likely to creep in with new versions. For reusability to be effective, document binding (contents pages, titles, footnotes, page numbers, etc.) should also occur automatically depending on the format chosen.

4.3 A Group Writing Model

Models for individual writing have been presented in chapter two (Flower & Hayes, 1981), along with some methods of coordinating group writing (Rimmershaw, 1992; Sharples et al., 1993). Collaborative writing, as mentioned before, is a complex task relating to the dynamics of the group, the coordination of people who may be geographically dispersed, the task of writing, and the identification and resolution of conflict during the process.

This section describes a model for collaborative writing, extending the model suggested by Flower and Hayes (1981) and Sharples and Pemberton (1990). This model allows for authors to work either synchronously or asynchronously, and in close proximity or at a distance. It identifies four main stages in the writing process for multiple authors, which are: *coordination*, *writing and annotation*, *consolidation*, and *negotiation*. These stages are iterative, continuing cyclically until the authors have all agreed on a final version of their work. Figure 14 below shows a pictorial representation of the model relating to two authors working on a document.

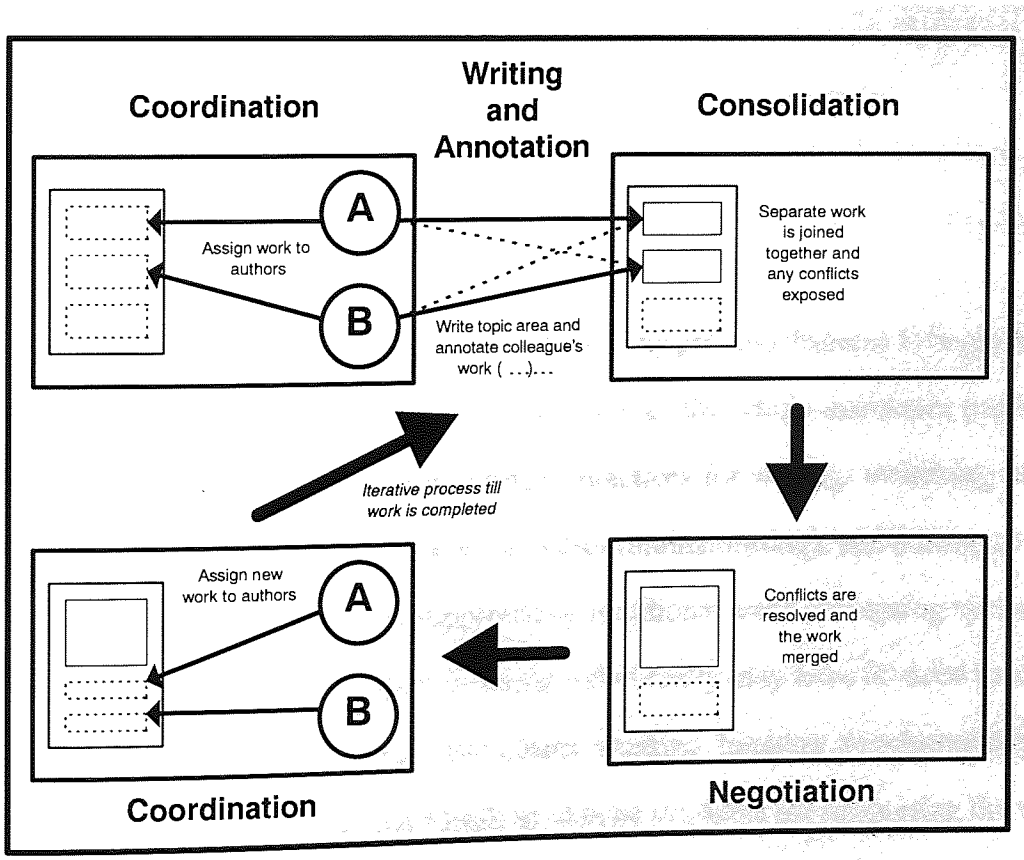

Figure 14: A model of the writing process between two authors.

4.3.1 Coordination

Coordination occurs both during and after the initial brainstorming and requirement specification stages of a writing project. Here the writers meet to set out the goals of the project, how the work should be divided amongst the authors, and determining what roles each author should take on. A significant amount of negotiation is also necessary at this stage so that authors will be satisfied with what they are required to do.

Depending on the authoring scenario, specific roles may be decided before some authors start on the project. In a newspaper authoring scenario, for example, the newspaper editor has total editorial control over the whole document. Responsibilities may be delegated to section editors, however, who would then have editorial control over their respective section (current affairs, sport, or arts, etc.). Journalists for the newspaper would only have authoring rights for the section (or column) they are reporting in. Similar hierarchical controls can be placed on other authoring scenarios, such as in a multi–user programming environment where a 'section' is a code module or, alternatively, in large, complex, and lengthy documentation projects, such as documenting safety–critical systems.

4.3.2 Writing and Annotation

4.3.2.1 Writing

In the writing stage the individual writers (or small writing groups) disperse to begin work on their designated topic areas. This stage is independent of the whole document production, with each author following their normal working practices for writing: switching between linear activities (the writing itself), creative activities (brainstorming), and editing activities as required. These activities are often supported by traditional word processing systems in a single–user system and, indeed, authors working individually may have no need to share or publicise their work till a later date. For teams working together synchronously on a collaborative document, however, other tools need to be available for supporting the writing activities, such as a shared text editor for writing and reviewing, and an ideas collator or electronic chalkboard for brainstorming.

4.3.2.2 Annotation

Annotation is used to handle the public sharing of comments between authors in a collaborative environment. Co–authors may write comments onto the shared document in order to draw the attention of the primary section author or editor to errors or suggested improvements with the text. Annotation comments can be added to a document in a distinctive way, such as through red–lining in some word processors, by attaching electronic Post–It ™ notes as with NoteCards (Trigg et al., 1986), by embedding spoken messages, or through a multi–column text layout for depicting the changes as with PREP (Neuwirth et al., 1990).

The use of annotation in a collaborative authoring environment is especially important for supporting three different activities:

- **Adding non–reserved comments to another author's text.**

  Non–reserved comments are comments which can be used to make suggestions or point out errors in a text to another author. The annotator is generally content to leave the original author to decide if and how they should take note of the comment. These comments may also be deleted by either the original annotator, or any of the authors it may have been addressed to. Alternatively, it may be left as an *aide memoire* for when the text is next edited.

- **Adding reserved comments to another author's text.**

  Reserved comments are comments which may be placed by an author to notify others of potential objections or conflicts to part of a text. Only the sender of the annotation has the right to remove a reserved comment, which would be done if the objection had been agreed with or negotiated on and can be withdrawn. If the comment cannot be agreed on informally between writers during the writing and annotation phase, it is transferred to the section authors for resolving during the negotiation stage, or demoted to the specific portion of the document it relates to for deciding on at a later date.

- **Notifying an author of previously consolidated work.**

  If a portion of work has been previously consolidated, and depending on the authoring scenario, it cannot then be easily changed. In the development of a contract, for example, it may be preferable not to change parts of the document once those parts have been agreed upon, in which case even the original author should not be able to amend the text. Annotations may thus be automatically added by the system over specific portions of text to show that those portions have already been consolidated, and that any further changes may not be added.

As well as annotations made directly to the text, other forms of messages may be sent between authors. Electronic mail systems can be used to send *directed messages*, as applied in Quilt (Leland et al., 1988), for sending private messages or notes between two authors. Alternatively *broadcast messages* can be used for sending messages to groups of authors, and can be used for managing coordination issues, for stating an author's intentions in order to increase group awareness, or as a means of using the system for simple teleconferencing.

4.3.2.3 Access Rights

As noted previously in this chapter, an author may not necessarily have the right to read or manipulate all parts of the document under preparation. This necessity to restrict access may be due to:

- coordination reasons, where the group decides to formally restrict access in order to avoid potential conflicts;
- security reasons, where parts of the document contains sensitive information; or
- performance reasons, where network communications traffic and the number of replicated copies of document node can be reduced.

As mentioned above, access rights can be either positive or negative, and weak or strong. Possible rights are:

- no access,
- read access,
- read and comment access,
- full access (read, comment, and write), and
- delegate access, allowing an author to grant his sets of rights (or a subset of them) to another author.

These access rights can also be linked to author roles, so that possible roles are:

- a ***reader***, with read access;
- an ***annotator***, with read and comment access;
- an ***author***, with full access; and
- an ***editor***, with read, comment, and delegate access.

While strict access rights may control coordination, especially with asynchronous working so that one author does not write in another's "space", there are advantages in enabling the group to read what each author is working on. Firstly, group awareness is improved between the authors, resulting in a smaller chance of writing cross–over conflicts. Secondly, authors may help improve the quality of the whole piece by giving suggestions to one another and, indeed, may use a colleague's work for stimulating their own ideas.

4.3.3 Consolidation

In the consolidation stage of the writing process, work done by the different authors in the project is merged together to form a single document. *Structural conflicts* occur when two or more authors write individual pieces of text for the same part of the document. If the work

assigned in the coordination stage has been clearly and simply defined then structural conflicts cannot arise.

When authors are permitted overlapping access rights to specific parts of a document structural conflicts can occur, although with good group awareness these conflicts can be prevented. Thus, if document changes are made to a single centralised document, then the system can use direct messages to inform the authors involved that a potential conflict may have arisen. Authors can also see graphically which authors have been assigned to various portions of text, and so possible conflicting areas can be identified in the coordination stage.

If two versions of a section have been written, generally as a result of the authors working asynchronously apart, then the system should make the work of the author with the most specific rights to the conflicting section the main text, and the other author's work becomes an annotation on that text as a reserved comment. The ensuing conflict is then carried forward to the negotiation stage.

Other conflicts which arise, as noted previously in the writing and annotation stage, are those of reserved comments by one author attached as an annotation to another's work. These conflicts are not structural, often referring to only a small portion of the text, and are generally resolved by direct messages and informal negotiation between the authors before the consolidation stage is reached. If the conflict remains, however, then the annotation is kept and carried forward to the negotiation stage, as with structural conflicts.

4.3.4 Negotiation

The negotiation stage of the writing process aims to diffuse any reserved comments or conflicts which have been identified and carried forward from the consolidation stage. In face–to–face proceedings, negotiation involves the ability to make comments or suggest changes, to argue for or against those changes, and if appropriate to vote on the changes. These same processes also need to be supported in a collaborative writing environment.

Many of the negotiation processes which occur can be dealt with informally in the writing and annotation stage through the use of direct messages and annotations. These informal

negotiations may also be used as historical references in a formal negotiation involving a neutral arbitrator or editorial board. As detailed in chapter two, much research has been conducted into computer–based negotiation and voting tools, like those used in Negotiation Support Systems (Holsapple et al., 1990), and which may also be used to help manage the negotiation process in collaborative writing.

To help explain their views and argue their points, authors and editors can use shared whiteboards and text editors to clarify their cases in the context of the document being prepared. Issue–based tools, consisting of issues, positions, and arguments, may also be used in a formal negotiation, as used in collaborative systems such as gIBIS (graphical Issue Based Information System; Conklin & Begeman, 1988). Once a negotiated settlement has been achieved, the settlement should be logged and the document updated so that discrepancies, such as annotated changes, are removed.

4.3.5 Coordination

Once conflicts have been resolved a new version of the working document is agreed upon. In some instances, it may be desired to freeze parts of the document, as with a legal contract, so that it cannot be changed by default at a later date. Another possibility is for particular conflicts to be "put on ice" and resolved later. This can have consequences on subsequent or future work, however, and proxies may need to be created so that subsequent work can reference the result of a negotiation can be referenced and eventually resolved once negotiations are concluded.

The next allocation of work is then made, based on the work done to date and the work still to be done. This can involve authors taking on editing or reviewer roles for previous work, or writing sections of the document which have yet to be completed. The authors next return to the writing and annotation stage of the writing process, which continues to iterate through the different stages until the authors are sufficiently satisfied with their work for it to be submitted for publishing. With continuously ongoing documentation projects, such as documentation for systems which are constantly being changed or improved, this closing stage may never be reached as the project does not come to an end.

4.4 Communication Flows between Writers

Collaborative work makes use of many varied and rich forms of communication between authors. A collaborative writing system needs to be able to handle these forms of communication. Four main types of communication have previously been identified for use in the writing process: *direct and broadcast messages*, *annotation/commenting*, *writing*, and *negotiation*. These modes of communication can be handled in a system by four output windows:

- **Document window**

  This window is based on a shared text editor, and allows the editing of document nodes. It has functionality for asynchronous and synchronous use, along with simple formatting controls (bold, italic, font style, etc.) which may be controlled through a "style sheet", thus allowing different authors to choose their preferred fonts and styles for the document.

- **Annotation window**

  The annotation window is a transparent window that sits on top of the document window and allows comments to be posted by different authors. The graphical appearance of the annotation window can be varied between a number of formats, such as a link icon pointing to the annotation, text written 'over' the document node's text, or a PostIt™ note placed over the document node.

- **Communications window**

  Based around current electronic mail systems, the communications window allows direct and broadcast messages to be sent between authors. If agreed by the authors, the text from these messages can be copied to the negotiation window as historical information in a negotiation.

- **Negotiation window**

  The negotiation window is based on a shared electronic chalkboard. It is designed primarily for synchronous use, and includes features for helping to manage negotiation

proceedings. It is on the negotiation window that work is coordinated, arguments made, and agreements decided on.

Figure 15 below depicts these four windows, and the interactions on them between two authors.

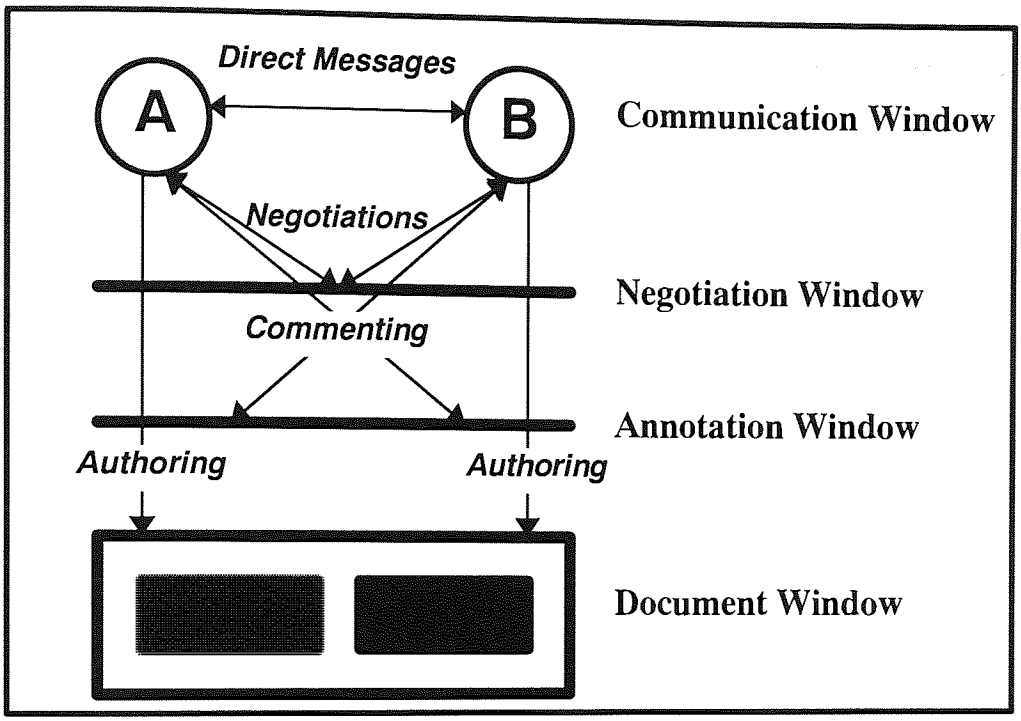

Figure 15: Communications windows in a collaborative writing environment, and the interactions between two authors.

Various levels of access is made to each of the windows during the different stages of the writing process.

In the coordination stage, the negotiation window is used for making and recording decisions regarding the goals of the project and the partitioning of work. During the writing and annotation stage, the document and annotation windows are used and the previous negotiation window frozen as a record of the last set of decisions made. All of these negotiation windows are kept, forming the basis of a history of the document development. New negotiation windows may also be created independently by any author, to be used as private scratch–pads for outlining ideas or plans. In the consolidation stage any unambiguous links between the document and annotation windows are merged into the document window, with any conflicts being transferred to the negotiation window.

The negotiation window, as used in the negotiation stage of the writing process, is the most involved of all the windows. Here the "negotiation window" actually refers to a hierarchy of windows, ranging from general to specific conflicts spanning the whole document, of which each window displays comments, positions, and arguments of the people involved in the negotiation.

The writing system should deal with general conflicts first and specific conflicts last, using a top–down approach to conflict resolution throughout the whole document. This approach is preferable as general comments are likely to affect larger portions of the document than specific comments, possibly causing specific comments to be automatically nullified. For example, if a general comment saying that section 3 of a document should be removed is agreed upon, then any specific comments relating to the text of section 3 become irrelevant. During the consolidation stage prior to this, however, a bottom–up approach is adopted so that as much as possible of the document can be merged together into a single version.

Throughout all the stages of the writing process the communication window is available, allowing authors to communicate directly with one another. This window could also be used by the system to transmit messages to specific authors, such as warning a group of authors that a structural conflict may potentially occur, or for querying an author as to whether a new participant has certain access rights to specific sections of the document.

4.5 Conclusion

This chapter describes the broad design of a computer–based writing system to aid authors work together in creating a joint document. It details the main issues that should be addressed by the system, and then describes the writing process between a group of people, noting that it is an iterative process consisting of a number of major stages: coordination, writing and annotation, consolidation, and negotiation. Finally the chapter presents a method of controlling the communication flows between authors, showing how different forms of communication are appropriate during different stages of the writing process.

The following chapter will expand on this chapter to examine the technical problems inherent in implementing groupware, focusing on the creation of a simple collaborative application.

Chapter 5

Collaborative Application Implementation Issues

5.1 Introduction

Software development of any kind is a complex process, with various methodologies developed to help the programming teams in their endeavour. Little work, however, has been done on methodologies for developing multi–user groupware applications, which have significant differences over their single–user counterparts.

This chapter will begin by looking at some of the main architectures which have been used for the development of multi–user applications, and the advantages and disadvantages of each. It will then present details of a groupware toolkit, called GroupKit (Roseman & Greenberg, 1995), which has been developed to aid in the production of collaborative applications.

Finally, the chapter will look at the ideas of *environmentally–aware software*, and the development of a simple prototype application to encompass the ideas of it, building on it to create a simple collaborative application.

5.2 Basic Collaborative Architectures

Building a collaborative application is totally different from that of building a single–user system. Collaborative systems, as well as having to deal with numerous organisational, sociological, and psychological factors, must also be developed so that they can be either transparently linked for group use, or otherwise be aware that they are part of a collaborative environment and 'know' how to communicate with other applications.

5.2.1 Group Awareness or Ignorance

The first element to decide when creating a collaborative application is whether it will be *aware* or *ignorant*. If an application is "aware", it knows about the possibility or presence of

more than one, potentially simultaneous, user. This can make the system more powerful and flexible, allowing for the introduction of various methods for cooperation and potentially reducing the need of additional communications channels between members. Such systems, however, can be regarded as more difficult and expensive to build, and are generally application–specific.

If an application is "ignorant" it acts as if there is only ever one user present. It is generally developed as a single–user program, subsequently being more general and requiring less forethought than with a multi–user program. As a result, however, the system is neither as flexible nor as powerful as its multi–user equivalent with a closed system stifling many varieties of collaboration that occur as a result of the ingenuity of the participating group members. Such systems also rely on other technologies for managing the group interactions and communications flows, generally through the use of shared views, such as with the Timbuktu™ (Farallon, 1988) application for the Apple Macintosh. These systems are inexpensive to implement and, while being limited in power, can be regarded as "stepping stones" to true collaboration aware systems (Johansen, 1989).

5.2.2 Application and Data Duplication or Centralisation

The issue as to whether to have a central application and centralised data, or have the application and data distributed and duplicated amongst the users, is one of the key implementation issues for a collaborative system. It is also a classic systems design issue in database design, consisting of a trade–off between data consistency and integrity when duplicated, and speed and accessibility when centralised.

In a collaborative system, where there is more than one user, *absolute* centralisation is impossible, as processes at least need to be running on individual machines. As a result, the main consideration with a collaborative system is not *whether* to duplicate, but *what* to duplicate.

The basic choice available is whether to duplicate the actual application and/or data, or duplicate the view of the data.

5.2.2.1 Duplicate application and/or duplicated data

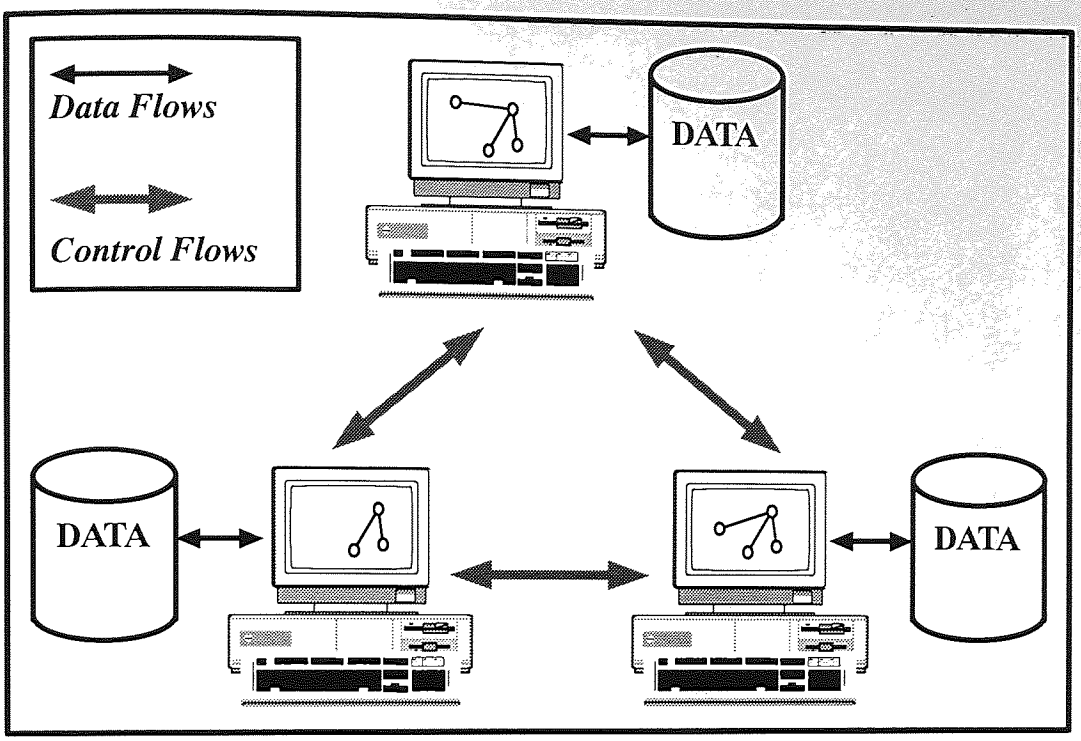

Figure 16: Duplicate applications and duplicate data.

When the application and data is duplicated, more than one process/data file is created. The multiple applications then communicate with one another, resolving conflicts and managing multiple copies of the data. This enables the system to be used over large distances, giving a faster local response, and allowing different sites to be operational at different times. The system needs to spend a lot of time dealing with synchronisation issues, however, and may lead to confusion from the users if their basic What–You–See–Is–What–I–See (WYSIWIS) assumption is broken, such that different users may be looking at slightly different versions of the document. Figure 16 above depicts an example of this architecture.

An alternative method to this, as shown in figure 17 below, is to duplicate the application, but keep a single centralised repository for the data. Such an approach could be implemented through a database system, where each user has a copy of the application to facilitate input to and output from the database. This method has the advantage of being able to make use of the database transaction and locking facilities, and also work over both large and small distances. Data may get out of synch and group awareness reduced, however, if large portions of the

document get "locked out" of sight of the participants. Nevertheless, such a method is particularly useful in systems requiring little data changing, such as in hypertext browsing.

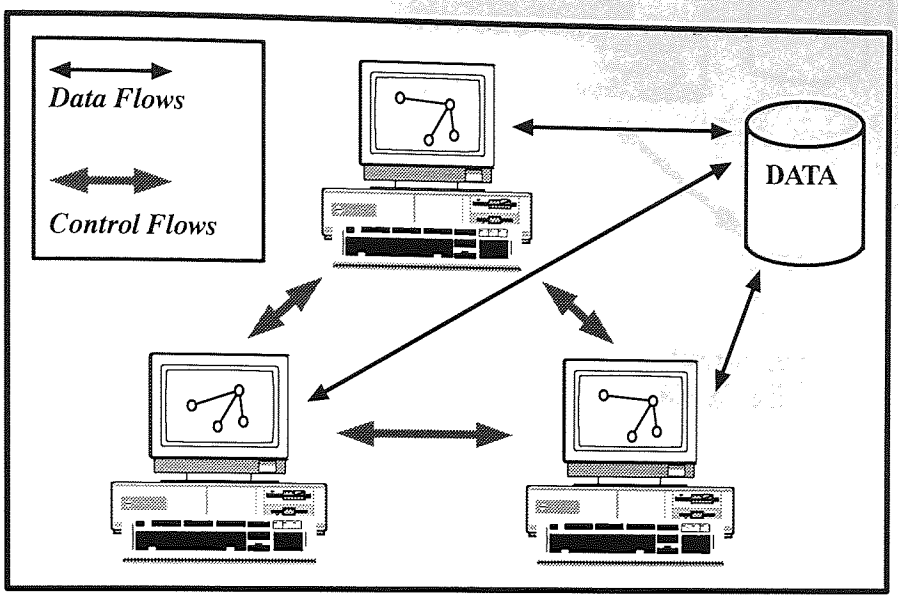

Figure 17: Duplicate applications with a single copy of the data.

5.2.2.2 Duplicate view

In systems where the view is duplicated, a central server application manages one set of data, but presents several independent views of the information to the users. The input from the users is coordinated or channelled through some method of *floor control* which determines which participant is able to write to the data at any time. Such an architecture is used by many of the shared whiteboards and editors, such as ShrEdit (Olson et al., 1992), and is most useful for small group teamworking on local area networks. As group size increases, however, the system can be overwhelmed due to heavy use in both input and output on the system.

Because views can be tailored for individual use, the same underlying data can be presented in different ways to different users, as shown in figure 18 below. Thus, an accountant can be looking at a set of data in terms of profitability, a marketing manager in terms of market share, and a production manager in terms of efficiency. This idea can also be expanded across international boundaries with a common document being automatically translated into different languages as required by the various participants (Wolfe, 1991).

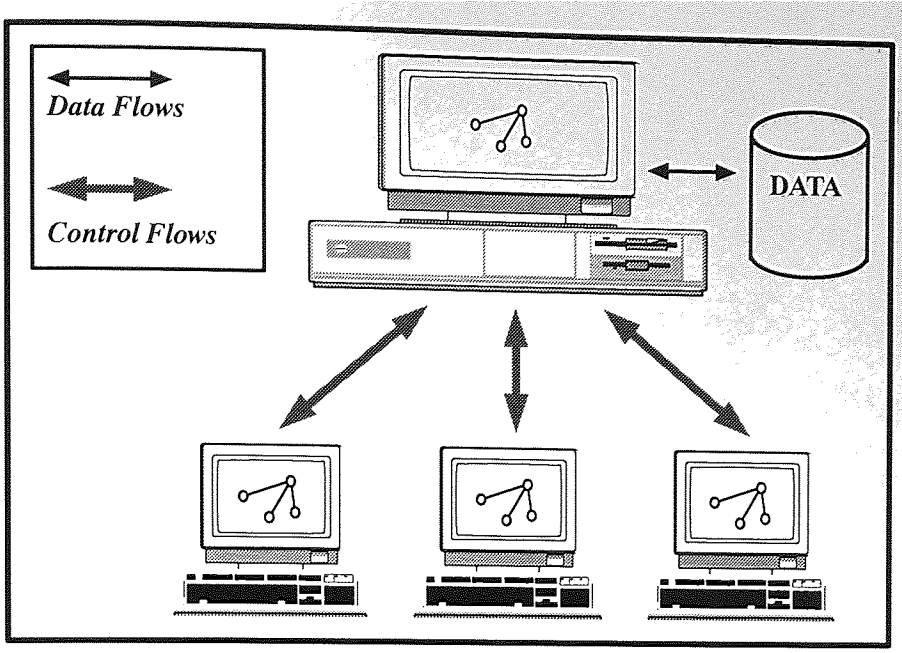

Figure 18: Duplicate views.

5.2.3 Floor Control

Floor control refers to a method of coordinating multiple inputs between numerous participants to result in a coherent output. As mentioned in the last sub–section, floor control is an important concern in groupware systems which use duplicate views for collaborating between participants. Without some form of floor control the input into an application becomes a nonsense. For example, two people typing simultaneously would have their inputs merged to form an inane sentence, or simultaneous attempts to move a single cursor could result in "cursor wars" (Greenberg, 1991).

Greenberg (1991) has identified a number of basic protocols for floor control. These are:

- **Free floor**

  Here the collaborators can enter input at any time, with floor control mediated through agreed social protocols or other means of communication (such as voice). Due to its nature, however, it is possible to accidentally have multiple inputs simultaneously, and can become confusing as group sizes increase. Using an analogy of chalk on a whiteboard, this protocol is equivalent to all participants holding onto the chalk, mutually agreeing to who should move it at the time.

- **Pre–emptive**

  Pre–emptive floor control occurs when any participant can 'take control' of the input stream from the current floor holder. Again using the chalk analogy, this method is equivalent to 'snatching the chalk' away from the person using it at that time. Such a method may be disorienting for the person currently in control, however, who suddenly finds the control taken from them.

- **Explicit release**

  With an explicit release protocol, the current floor holder must explicitly relinquish his control over the input. Any of the other participants may then claim the floor. This method is equivalent to the floor holder setting down the chalk, where it may then be picked up by another.

- **First in, first out queue with explicit release**

  This protocol is similar to the explicit release protocol, with the only difference being that, instead of *any* participant being able to claim the input once it has been relinquished, the person who *next asked for it* is given it. Thus, this is the same as having a line of participants, with the chalk being handed from one to the next. One problem with this method is that people who asked for the chalk may not actually want it when it is finally given to them as the issue has already passed. Also, a participant near the end of the queue may write something which nullifies the use of the chalk by the collaborators in front of him.

- **Central moderator**

  With the central moderator protocol, a moderator oversees all activity and decides who should hold the floor. The moderator gives and relinquishes control of the floor, usually as a result of monitoring requests for control by other participants. Using the chalk analogy, this is equivalent to one person holding the chalk, lending it as appropriate to other participants asking for it. Such a protocol could be used by a negotiator or arbitrator in order to try and give a balanced view of an argument.

- **Pause detection**

  This method is similar to the explicit release protocol, except the release is *implied* through a suitable pause of activity by the floor holder. Thus, if a person stops writing with the chalk for a while, another participant may take it from them and start writing instead. As a result, this method has the potential of disorienting the floor holder in a similar way to the pre–emptive protocol, especially if the pause was simply to collect one's thoughts.

Each of these forms of floor control may be appropriate in particular situations or to specific groups. For example, a small group may prefer to use a free floor protocol, relying on social protocols to organise the usage. A brainstorming situation may also tend to use a free floor protocol. A decision–making process, however, might use a central moderator or first in, first out, explicit release protocol, so that each participant has an option of airing their views.

All the methods of floor control rely on some extent to social protocols. Thus, the explicit release protocol assumes that any one participant will not "hog" the use of the input stream, while the free floor protocol assumes that a participant will not type something while another is doing so. If these social protocols fail and one floor control method is found to be inadequate, participants should be able to adopt a different method. Also, as group sizes increase, so the reliance on social protocols (along with other communication), also increases.

The need to rely on some floor control protocol other than the *free floor* protocol depends on the extent of the *granularity* of the collaboration. That is, two participants may be regarded as either working on two separate parts of a document (where free floor control is used implicitly, as there can be no 'collisions' between the participants), or in close proximity (where floor control will play an important part of the collaboration). Depending on the proximity of the two participants to each others' work, floor control will either play a major role or have no impact on the collaboration. In terms of a collaborative writing project, the granularity of this proximity could be based on the structural components of the document. Floor control could thus be used when the participants intersect with their work at, for example, the section level, paragraph level, or sentence level. Other structural components, such as a list, a picture, or a header, could also be used for defining the granularity.

5.2.4 View Control

In a collaborative environment the method of view control, that is the architecture for replicating or displaying views of the project on multiple screens, is of great importance. There are a number of different approaches to view control, namely:

- **Multiple Displays**

  With a multiple display architecture, the centralised application client knows of what displays are connected to it and manages the input and output to those displays. Inputs are time–stamped and ordered by the communications channels between client and server. Users can join the collaboration simply by connecting their display server to the application client, or leave by breaking the connection to the client. The client application, however, needs to be designed for collaborative work as it determines what should be shown on each of the connected displays.

- **Shared Windows**

  A shared window architecture consists of software which transmits the output of one central display server onto a number of other display servers. This could be through either intercepting any output commands on the central server and sending the same commands to all the other connected servers, or through transmitting bitmap images of the central display to the other displays. The former method is the most economical with bandwidth, but requires all connected machines to accept the same output primitives.

  To join the collaboration, each user connects to the central server which contains the software for output duplication. To leave, the user simply breaks the connection with the server, using their own localised server for handling the display. When connected, each display portrays exactly the same view as the others. While the application client need not be designed for collaborative use, a disadvantage is that the participants can only view the same portion of text. Thus users are unable to work on disparate parts of the document at the same time. Nevertheless, this method is useful for work requiring close collaboration, and is the view method used in Timbuktu™ (Farallon, 1988).

- **View Manager**

  A view manager is placed between the application clients and display servers, and implements a 'view policy' depending on the applications connected to it. Such a policy can range from duplicate views for use in non–collaborative applications, (as used by shared windows above), to periodic or selective updates of the display for applications written for collaborative use.

  In use, an application writes not to the output display, but instead to the view manager which in turn writes to the display. A view manager is the most versatile method of collaborating between applications but, because it is called between the application and display, may itself become a communication or processing bottle–neck, needing to be replicated as a process itself.

5.2.5 Process Control

In most of the current generation of computers, especially in systems with a windowing system of some kind, applications are written in an *event–driven* style, as in:

```
while (always) do {
  <get next event>
  <process event>
}
```

That is, the core of the application waits until some event occurs (such as the user clicking on a button, or typing some text). Once the event is received, a routine is then called to process it. This means that the application waits for some action from the user, deals with it, and then waits for the next action by the user.

In a collaborative environment, however, this event–driven style has a basic failing. That is, if two users are using the system, they will both be generating events for the application to handle. If one user chooses to perform a time–consuming task, they are generally prepared to wait before performing another action. Other users on the system, however, would suddenly find that their actions have no response, and not know why. These users can quickly get frustrated with the system and potentially lead to it not being accepted by the group.

To deal with this problem, the collaborative application could use a number of methods:

- **Wait States**

  The application could display an hourglass or "please wait" cursor when doing a process–intensive task. This is the same as what single–user applications generally do, and may be acceptable in a collaborative system where all the users are working in close collaboration, seeing the input which causes the delay, and also having a stake in seeing the result.

- **Multi–Threading**

  Within a time–consuming process, a form of non pre–emptive multi–processing could be used. That is, the process pauses, temporarily handing control over to the multi–thread controller. Other processes are then given an opportunity to run before the controller hands back to the lengthy process. Such a system, however, relies on time–consuming processes passing control over to the multi–thread controller at regular intervals.

- **Idle Processing**

  Idle processing techniques are common in single–user event–driven applications. Here, most of the processing work is accomplished when the users are not actually implementing a command. For example, during the time required for a user to read a message the system can be processing a previous command. In a group environment, however, there is less opportunity for *idle times* as one user might be reading, while another is typing some text, while another is processing a command.

- **Multi–Processing**

  Here, lengthy processes are forked off to a separate task to handle the long–duration processing. This task then runs independently of the main application, freeing control for other users' inputs and commands. The user who instantiated the task will need a flag or checkpoint added so that, once the lengthy process is complete, it can be returned at the point where the original user called it.

- **Multiple Event Loops**

  With multiple event loops, the process which is triggered as a result of the first event itself has an event loop, checking for high–priority events from other users. Thus, the process code could look like:

```
/* routine for doing a processor-intensive task */
while (task not finished) do {
  <see if there are any high-priority events>
  <if there are any high-priority events,deal with them>
  <complete some more of the processor-intensive task>
}
```

  This method works well if most of the generated events can be processed quickly, and where there are not large numbers of events generated. The system breaks down, however, if another high–priority event is processor–intensive, as the first lengthy task is paused while another time–consuming task is performed.

- **Social Controls**

  As well as technical options as detailed above, *social controls* may also be used in a collaborative working environment. Such controls could include a person formally or informally asking the other members whether it is acceptable for him to do a specific action, or group norms requiring potentially time–consuming processes to be left to run during 'quiet' periods on the system. Social controls are particularly important where the consequences impact on other users and, indeed, certain processes may not be able to be performed if a group member has not granted access rights to their section. For example, a member may want to perform a global search and replace on a word, deciding that another word is more apt in its place. Whilst the member might not have been working in close proximity to other members, the global change will affect all participants' work. The member must first ensure that it is acceptable to proceed with the change, or at least have it form a new version which may then be negotiated and consolidated on at a later stage.

A collaborative application could use any of these methods for different users. Thus, a wait state method could be used for the person who asked for the time–consuming operation with multi–threading used for other users' inputs. Depending on the situation, a subordinate

member might also always be required to ask their superiors if they can make a change or execute a process.

5.3 GroupKit – A Collaborative Development Environment

GroupKit is a toolkit for developing collaborative applications for synchronous and distributed computer–based conferencing. Work on the toolkit first started in 1991 (Roseman & Greenberg, 1992), and has since been continuously developed at the University of Calgary by Mark Roseman and Saul Greenberg. Their belief is that:

> *"A developer using a well–designed toolkit should find it only slightly harder to program useable groupware systems when compared to the effort required to program an equivalent single–user system" (Roseman & Greenberg, 1995).*

As a result, the toolkit has been designed around a set of design requirements which take into account both user and programmer considerations. These requirements are:

- ***Support multi–user actions and awareness of others over a shared visual work surface.***

  This may be achieved through means such as using telepointers for gesturing, having graphical annotations for notes, and multi–user scrollbars for identifying what people are viewing.

- ***Provide support for structuring group processes, but do so in a flexible enough fashion to accommodate the diverse needs of different groups.***

  This means that the toolkit is flexible enough to support the needs and styles of different groups, such as different methods of floor control and session management, and how to treat access and latecomers to a conference.

- ***Integrate groupware with conventional ways of doing work.***

  The system should allow users to work in familiar ways, and not pose a barrier to "traditional" methods of doing work. Thus external resources should also be available, such as telephone and video links.

- ***Provide technical support to deal with multiple distributed processes.***

  The toolkit should simplify the process of creation, interconnection, and closure of the multiple processes required to communicate over a network. Thus the toolkit should handle session management and process creation, along with the ability of locating other processes and doing multicast abstractions.

- ***Provide support for shared data.***

  The toolkit should be able to easily share its common data, with provisions to detect and take action when the data has changed, and concurrency control methods to keep the data consistent. This can be accomplished through the use of shared environments, data serialisation, and binding callbacks to environment events.

- ***Provide support for an extensible shared graphics model.***

  The toolkit should contain graphics primitives useful for shared working, which may be manipulated and extended in specific applications. The view of the object should also be separate from its data representation, allowing for different graphical views of the same object by different users.

5.3.1 GroupKit Development Architecture

GroupKit contains a number of features and facilities to aid in the development of a collaborative application. These are:

- a runtime infrastructure for managing the creation, interconnection, and distributed process communications that comprise a collaborative conference;
- session managers to aid in the creation and management of collaborative meetings;
- groupware programming abstractions for distributed process control and data sharing;
- user interface widgets of use in groupware applications.

5.3.1.1 Runtime infrastructure

The runtime infrastructure of GroupKit controls the core communications issues of a collaborative application. This includes process manipulation, socket handling, and message

and command multicasting. The infrastructure consists of a range of processes spawned across a number of machines. Figure 19 below shows an example of the processes running when two people are collaborating using two conference applications, 'A' and 'B'.

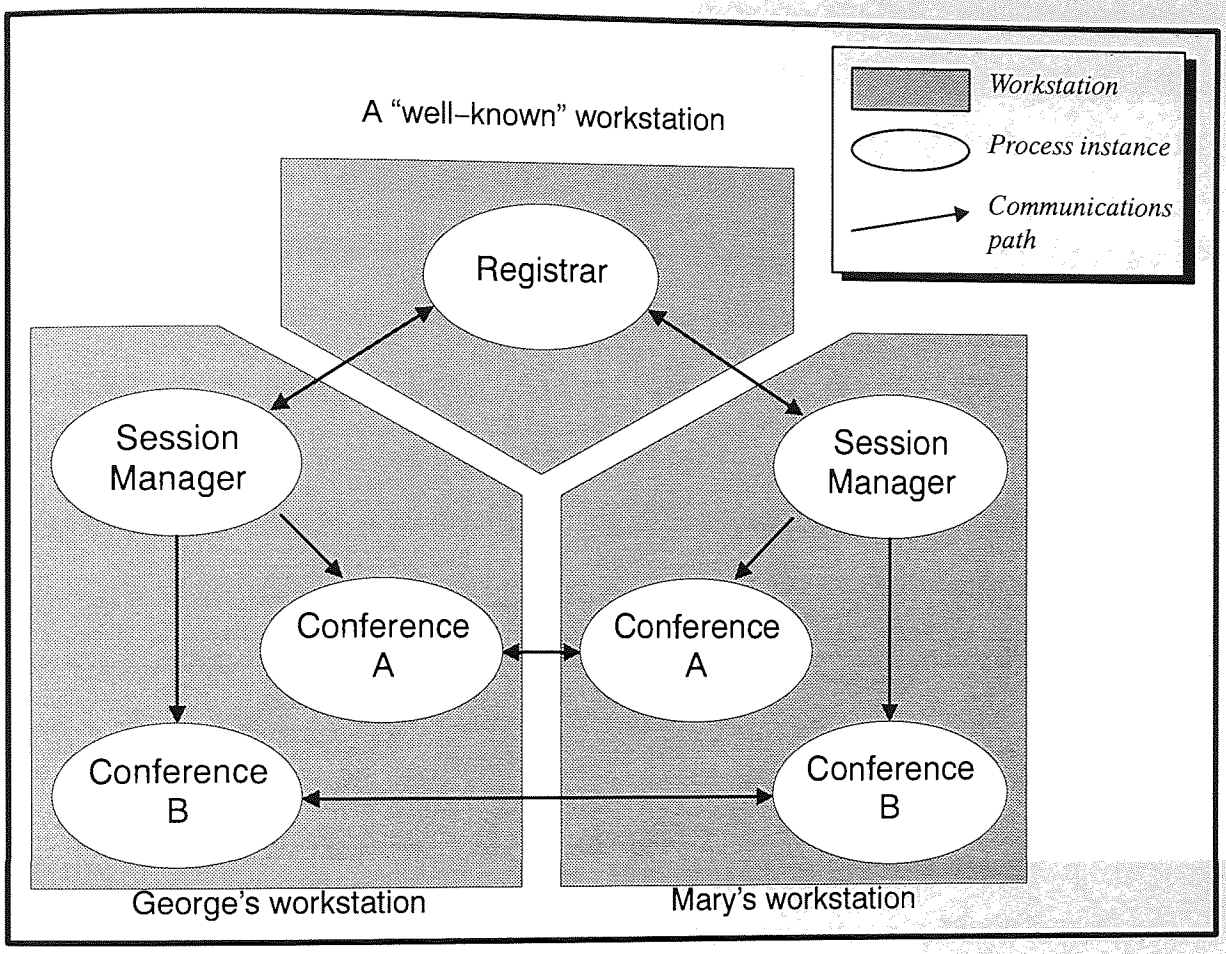

Figure 19: GroupKit's runtime process model.

Adapted from: M. Roseman & S. Greenberg (1995), *Building Real Time Groupware with GroupKit, A Groupware Toolkit* p.9

From this figure, it can be seen that there are three main types of process used by GroupKit in a collaborative environment: *a central registrar, session managers*, and *conference applications*.

The Registrar is a centralised process which runs on a workstation that other machines know how to reach. It maintains a list of all the conferences running, and the users in each conference. Thus, it serves as an initial contact point for locating existing conference application processes.

The session manager is a replicated process, one for each group member, that controls the policies used by the group for how users create, join, leave, and delete conferences. It

communicates with other session managers only through changes which have been made to the central Registrar, so a session manager needs only to monitor the data from one specific process. Different session managers may be developed depending on the policy required by the group so that, for example, a manager can implement an open policy where group members can create, join, or leave a conference at any time, or implement a closed policy where members are unable to join a conference unless asked to do so.

A conference applications is the main collaborative application program developed to provide some kind of functionality for the group members. It is a replicated process, with a copy running on each participants' machine. Together, all the instances of the conference process running on each machine constitutes a conference session, which is managed by the session manager. Conference processes may ask for and send messages and commands to other processes in the session, enabling direct communication links to be available between conference processes. Examples of conference applications are a shared chalkboard, a voting tool, a shared editor, or a game such as tic–tac–toe.

5.3.1.2 Session management

As mentioned above, GroupKit decouples the issue of session management from the collaborative application through individual session managers, thus enabling the system to be flexible enough to manage a variety of group processes in different groups.

It is the session manager that controls the policy for creating, joining, leaving, and deleting a collaborative session. These policies may be either implied or specific. Thus, if a person leaves a conference, having been the last person in it, then one policy may imply that the conference should now be deleted. Alternatively, for a person to join a conference, the session manager could explicitly ask the current participants if this is acceptable.

Cockburn and Greenberg (1993) have noted that one of the obstacles to computer–supported collaborative work is the difficulty in starting a groupware session. This could be a usability problem where the system is difficult to start, or a social problem where the policies provided by the system are inappropriate to the group. Thus it is important to have a flexible session

management system which is tuned to the needs and collaboration patterns of the target user group.

5.3.1.3 Groupware programming abstractions

GroupKit provides a set of programming abstractions to help application developers write conference applications that interact correctly with other distributed processes in a collaborative session. These abstractions are:

- **Multicast Remote Procedure Calls**

  Multicast remote procedure calls (RPCs) allow developers to communicate, share information, and trigger program commands between replicated application processes running in a collaborative session. This is based on the Tcl–DP mechanism (Smith et al., 1993) for handling remote calls, but is abstracted further so that users are simply referred to via unique user IDs. Table 3 below describes the main RPC commands as used by GroupKit.

`gk_toAll`	Multicast a procedure to all processes in the conference session.
`gk_toOthers`	Multicast a procedure to all processes in the conference session, except the local user instantiating the call.
`gk_toUserNum`	Send a procedure to a specific process, as identified by its unique user ID.
`gk_serialize`	Multicast a procedure to all processes in the conference session, but make sure they are serialized so as to be received in a specific order.

Table 3: The abstracted RPC calls available in GroupKit.

In GroupKit, the RPC calls do not wait for a response, the calling program simply continuing execution. This ensures that applications are not 'frozen' as a result of network latency problems or crashes on remote machines. A message system of asking for information, and then receiving it in another event, is preferred instead. There is, however, a switch that may be placed on the `gk_toUserNum` command if the application does require an immediate response from the remote process.

- **Events**

As mentioned above, GroupKit applications generally use a message system for communicating between processes. This is achieved through an event mechanism, similar to the way events are generated for user actions in window-based systems. As well as customisable events, a number of pre-defined events are available in GroupKit. These are described in table 4 below.

`newUserArrived`	A new member has arrived in the conference session.
`userDeleted`	A person has just left the conference session.
`updateEntrant`	A late entrant to the conference session needs updating.

Table 4: Events automatically generated in GroupKit.

An event consists of an event type, (such as 'newUserArrived' or 'userEnteredNode'), and a set of attribute/value pairs that provide varying pieces of information depending on the event. Events are trapped and handled through *bindings*, which call a particular command whenever its matching event name occurs. For example, to send a greeting to a user who has just joined a conference, the following code fragment could be used:

```
gk_bind newUserArrived {
  set my_name [users local.username]
  set new_user_name [users remote.%U.username]
  set msg "$my_name says WELCOME! to $new_user_name"
  gk_toUserNum %U PrintMessage $msg
}
```

Thus if user 'Mary' joins a conference session already consisting of users 'George' and 'Dave', she will be sent two messages saying: "George says WELCOME! to Mary" and "Dave says WELCOME! to Mary".

- **Environments**

Environments are data structures consisting of *keys* and *values* which can store any level of data in a hierarchical tree fashion. As well as being a data repository, environments are *active* in that events can be triggered when the environment variables are created, deleted, or changed. These events can be automatically multicast over all the conference processes

in the session, so that the data is shared between the processes and data integrity maintained.

Environments are also an effective method of supporting the model–view–controller model as used in Smalltalk (Krasner and Pope, 1988). Thus the data is abstracted from the view so that various types of view may be created from the same underlying data. This is of special relevance for personalising collaborative applications, where different participants may choose to have different individual views of the same data.

5.3.1.4 User interface widgets

Finally, GroupKit provides a set of user interface widgets which may be used by collaborative applications. These widgets are based around three areas of functionality: *participant status*, *telepointers*, and *location awareness*.

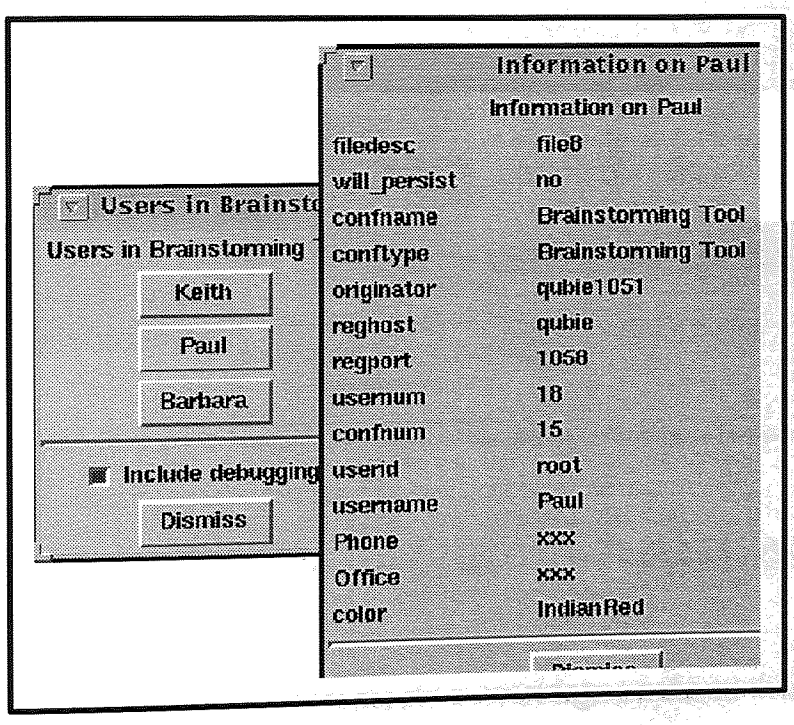

Figure 20: The Participants widget.

Participant status refers to the ability of members to "see" other group members as they join and leave conferences, along with some details about them. GroupKit uses a 'participants widget', as shown in figure 20 above, to achieve this. This widget dynamically displays a list of the current group members, with the ability of getting further details by clicking on a group member's name.

Gesturing accounts for approximately 35% of a group's activities when working together over a shared work surface (Tang, 1991) and is a rich communications medium. By gesturing, a group member can draw attention to a particular object, show relations between objects, and suggest what they intend to do next. GroupKit uses telepointers (also called "multiple cursors") to implement gesturing, where bitmapped pointers are created and attached to the underlying widgets in an application.

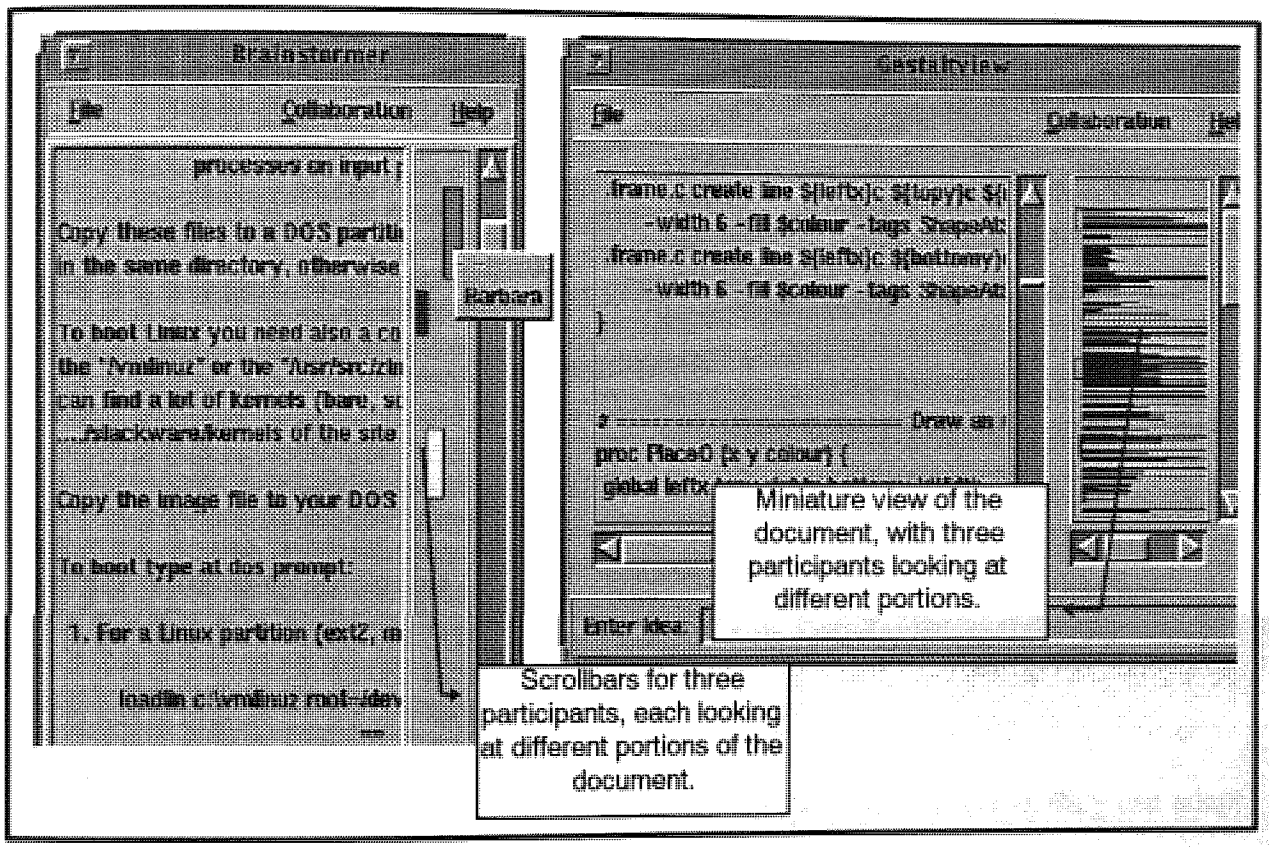

Figure 21: The multi-user scrollbar and gestalt viewer widgets in GroupKit.

Finally, location awareness refers to *where* people are working on a shared work surface. That is, different users may be looking at different portions of a document at the same time, and knowing *where* people are working can give an indication to *what* they are working on, thus helping improve work coordination. GroupKit uses a multi-user scrollbar to depict location awareness, with one scrollbar being added for each participant. The system also incorporates a gestalt viewer, which depicts a miniature view of the document along with coloured boxes depicting what each user is currently viewing. Both these widgets are depicted in figure 21 above.

5.4 "Environmentally-aware" Software

Environmentally-aware software applications are programs which know about the world in which they operate. Such programs adapt depending on what other programs are present, and operate so as to always remain functional to the best of their ability. This section describes a simple application of an environmentally-aware program, and how the concept can be expanded further.

5.4.1 Tic-Tac-Toe Example

A simple example of environmentally-aware applications can be depicted using a tic-tac-toe game. Tic-tac-toe is probably one of the most basic of collaborative applications, with two players using a turn-taking floor control policy to place their tiles on a simple grid and try to 'win' the game. In a basic programming implementation of this game, a conference application could be written which consists of all the aspects present in the game – namely the ability to place 'X' and 'O' tiles, and an understanding of the rules and turn-taking protocol of the game. Two conference applications would need to be running, one to play 'X' and one to play 'O' in order for the program to run.

If the conference applications are made environmentally-aware, however, they can adapt depending on the number of conference applications ('people') running in the session. Thus the state of the application changes dynamically as people enter and leave the world.

5.4.1.1 Tic-tac-toe world with one person

The system contains the functionality required to play the game. It knows how to play 'X', 'O' and mediate for the rules, and takes on all of these roles. The system, therefore, does not "*wait for the other player to connect*" but instead operates the whole system. Such a case might be present if two people wish to use a facility, but only have a single resource for it. Here, there is only one application running, but there could be multiple users of that application.

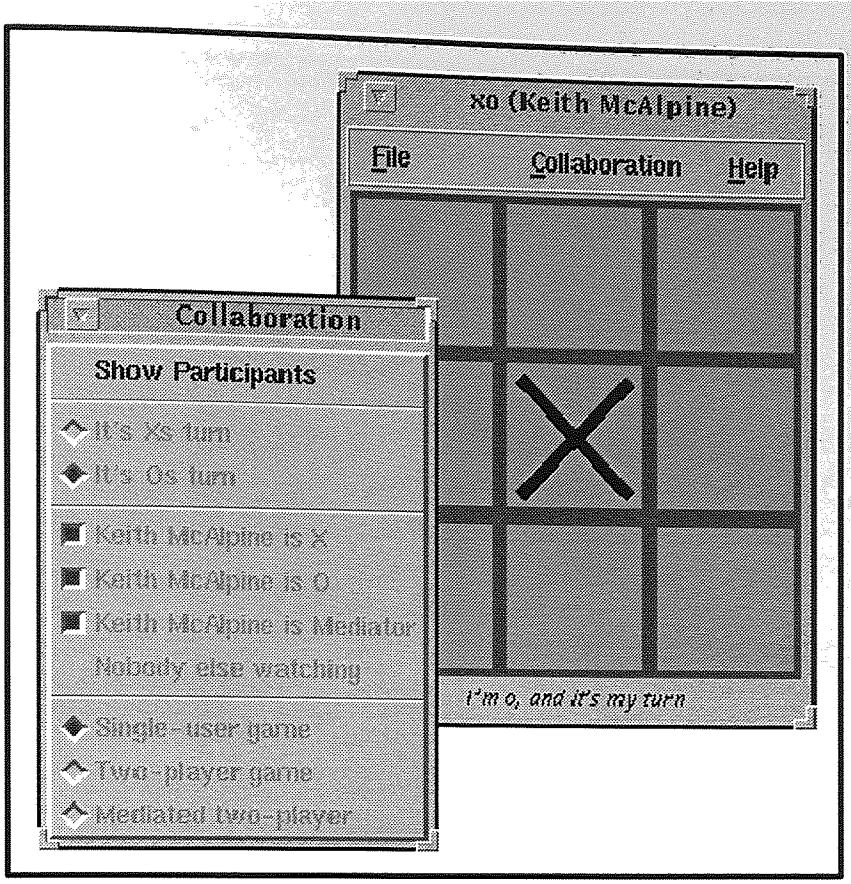

Figure 22: Tic–tac–toe game with 1 conference application running.

In the figure above, user 'Keith McAlpine' is the only person in world, and thus takes on all the roles of 'X', 'O' and mediator.

5.4.1.2 Tic–tac–toe world with two people

If another person joins the world, both applications adapt. From each being a 'single–user' application, both programs now talk to one another so that one application plays 'X' while the other plays 'O'. The figure below shows an example of this, where user 'Keith McAlpine' has been joined by user 'Paul Golder'.

The figure shows both conference applications running and the collaboration menu for each. The collaboration menus reflect the state of the world, showing who is present and what role they take on. The user 'Paul Golder' has thus taken over the role of 'O' from user 'Keith McAlpine'.

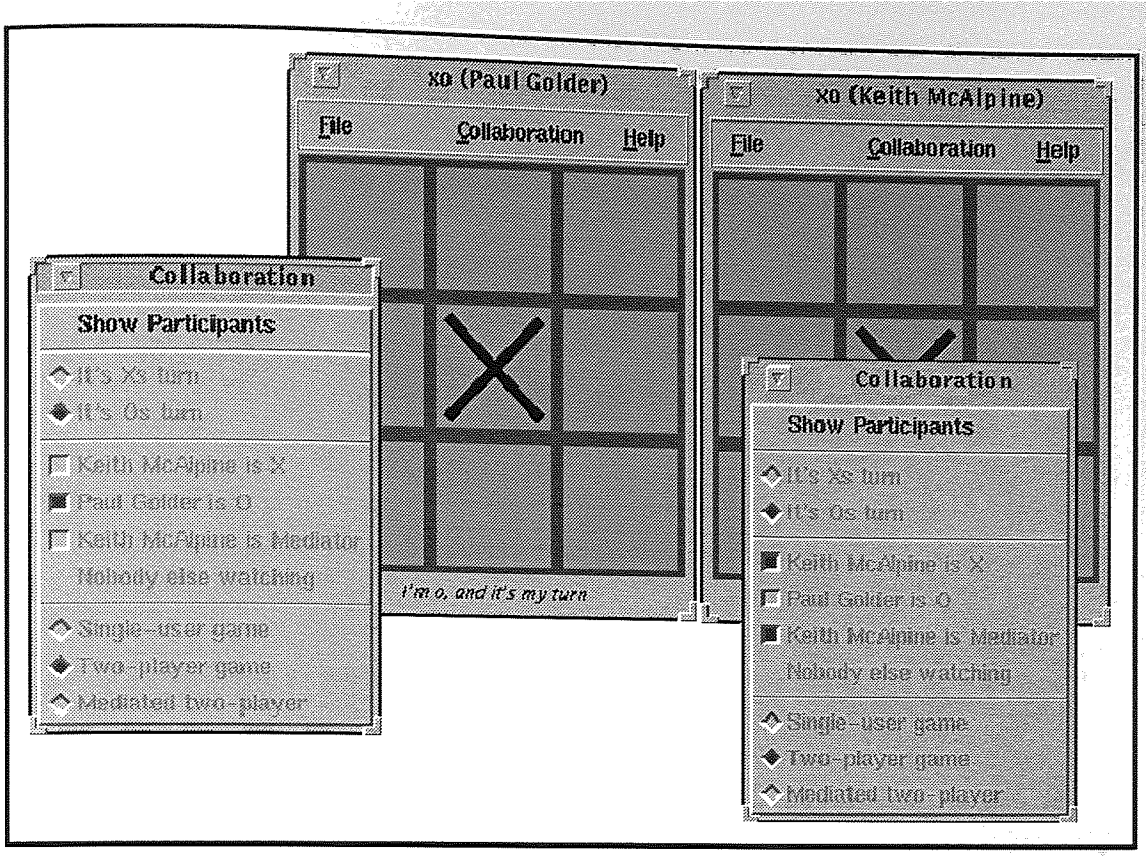

Figure 23: Tic–tac–toe game with 2 conference applications running.

5.4.1.3 Tic–tac–toe world with three people

When another person joins, the world again adapts. The 'X' and 'O' roles of the original members remain the same, but the new member takes over the job of mediating the game. That is, this member is in charge of the rules for the game, determining whose turn it is, where a player can place a tile, and whether the game has finished. The figure below shows this, with user 'Barbara Kriedte' joining and taking over the role of mediator. A typical series of messages as the result of a player clicking in a square is:

```
Player X: Request to place tile on grid x,y
Mediator: Receives request from Player X
          Checks game board to see if this is OK
          If OK, Sends message to X to draw the tile
          Updates game board data structure
          Check for game over condition
Player X: Receives request to draw tile at x,y
          Updates view to draw tile at x,y
```

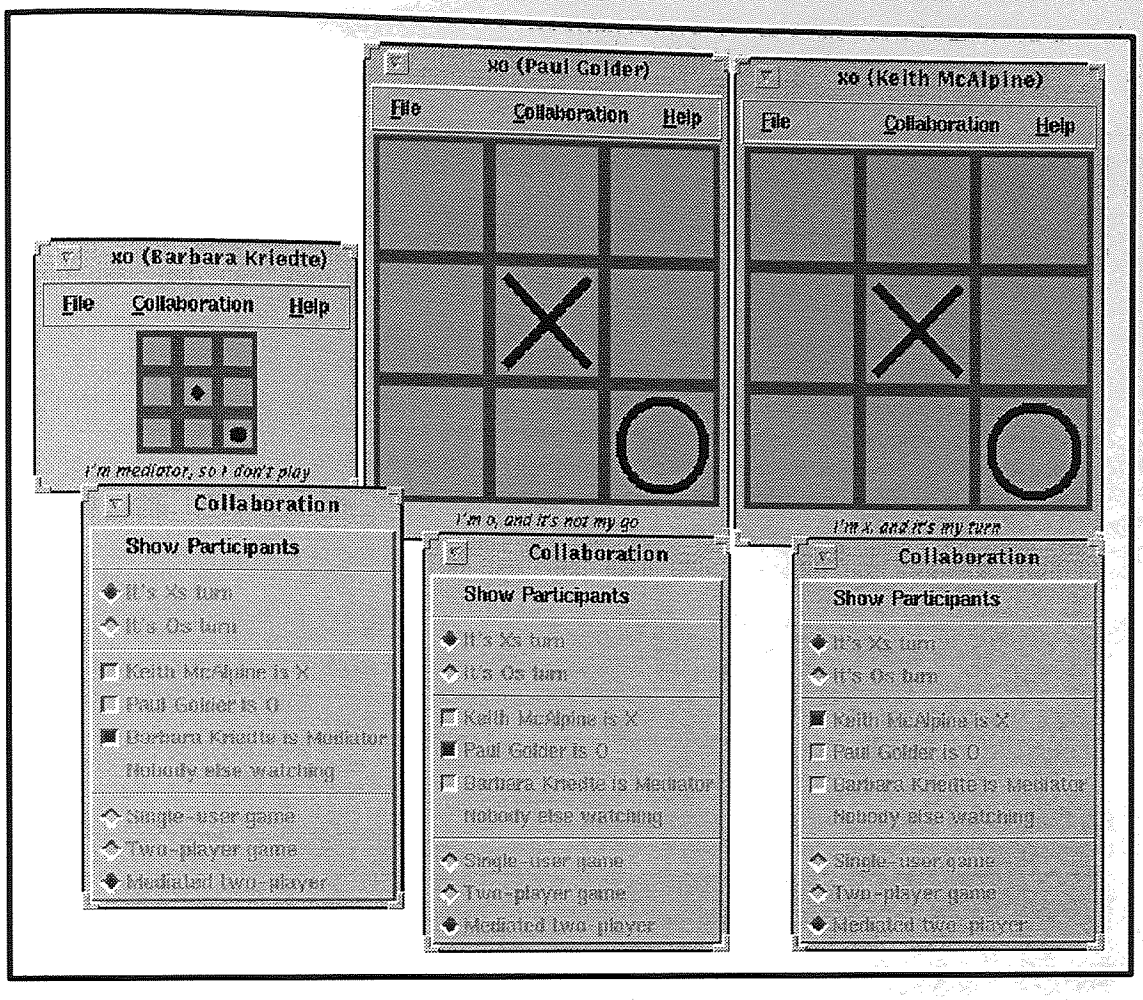

Figure 24: Tic–tac–toe game with 3 conference applications running.

From this message segment, it can be seen that the mediator is responsible for the underlying game data structure, with the individual 'X' and 'O' players responsible for their views of the data structure. Each person is thus responsible for their own 'expertise' – the mediator for the game state, player X for drawing an 'X' and its user–interface, and player O for drawing an 'O' and its user–interface. By using this distribution, any player can create specialised effects for their role. For example, the mediator could 'change the rules' and allow 'X' to overwrite 'O', or player 'O' could change his drawing expertise to draw something other than an 'O'!

This concept is similar to that used in object–oriented programming. Initially all objects have a basic capability so that they can function. As more objects become introduced, so they can inherit the basic capabilities of the original object, but also add further to those capabilities through specialisation. The interface remains the same, however, so that the objects can still work with one another.

5.4.1.4 Tic–tac–toe world with more than three people

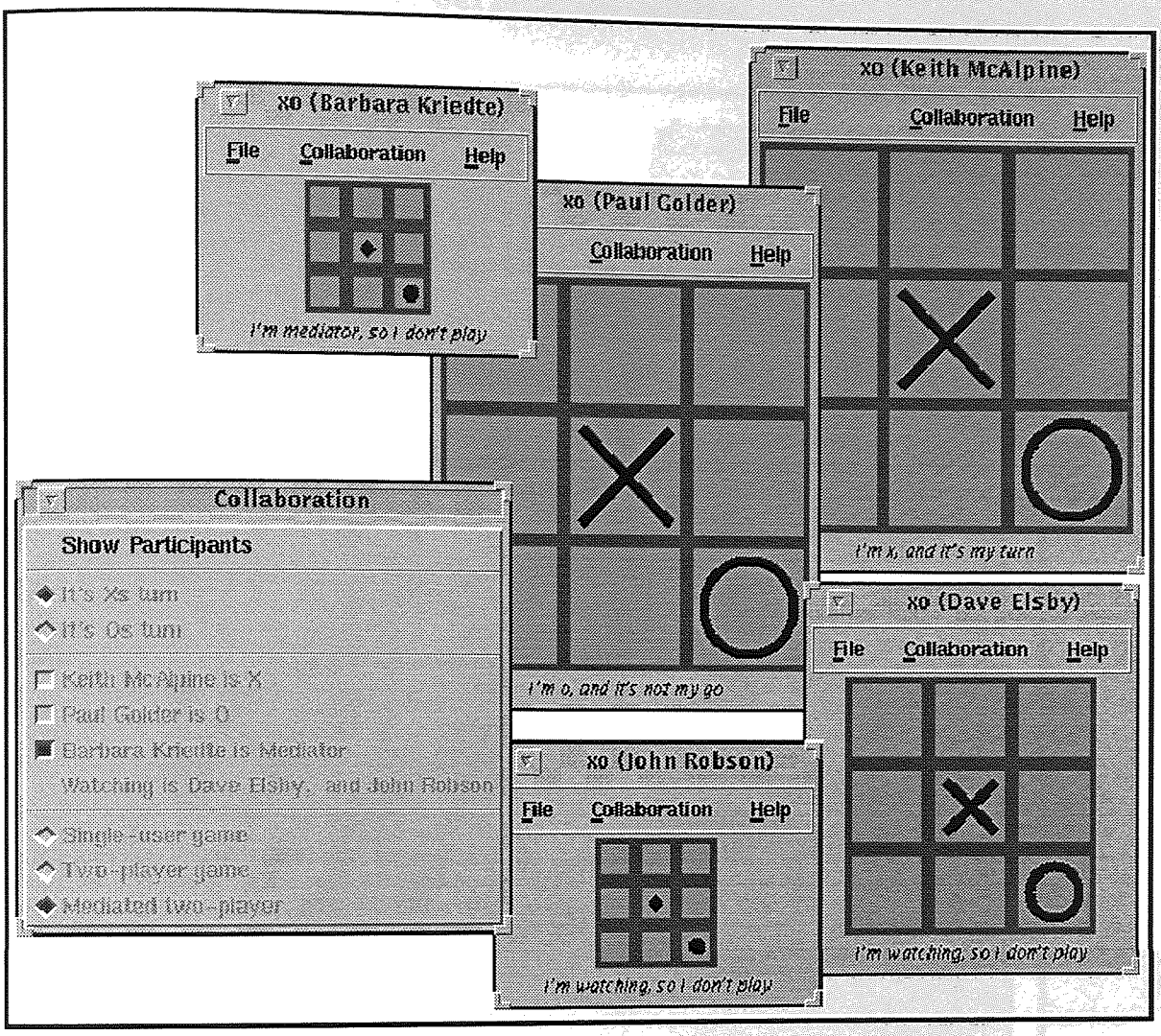

Figure 25: Tic–tac–toe game with 5 conference applications running.

As more people join the world, so the applications present in the world continue to adapt. The figure above shows the world with two additional entrants, users 'Dave Elsby' and 'John Robson'. In this case, the only role they can take is that of watching the game in progress. Each user can adjust their individual view of the world (represented by the size of the board in this example) and therefore allows for customisation and tailoring.

5.4.1.5 Tic–tac–toe world with people leaving

As people depart from the world, so all the participants in the world need to adapt to these changes. Figure 26 above shows an example of this where two users, 'Keith McAlpine' (previously acting as 'X') and 'Barbara Kriedte' (previously acting as mediator) have left the

world. The applications left need to adjust for this loss, and so those users who had previously been watching now take on the roles of 'X' and mediator.

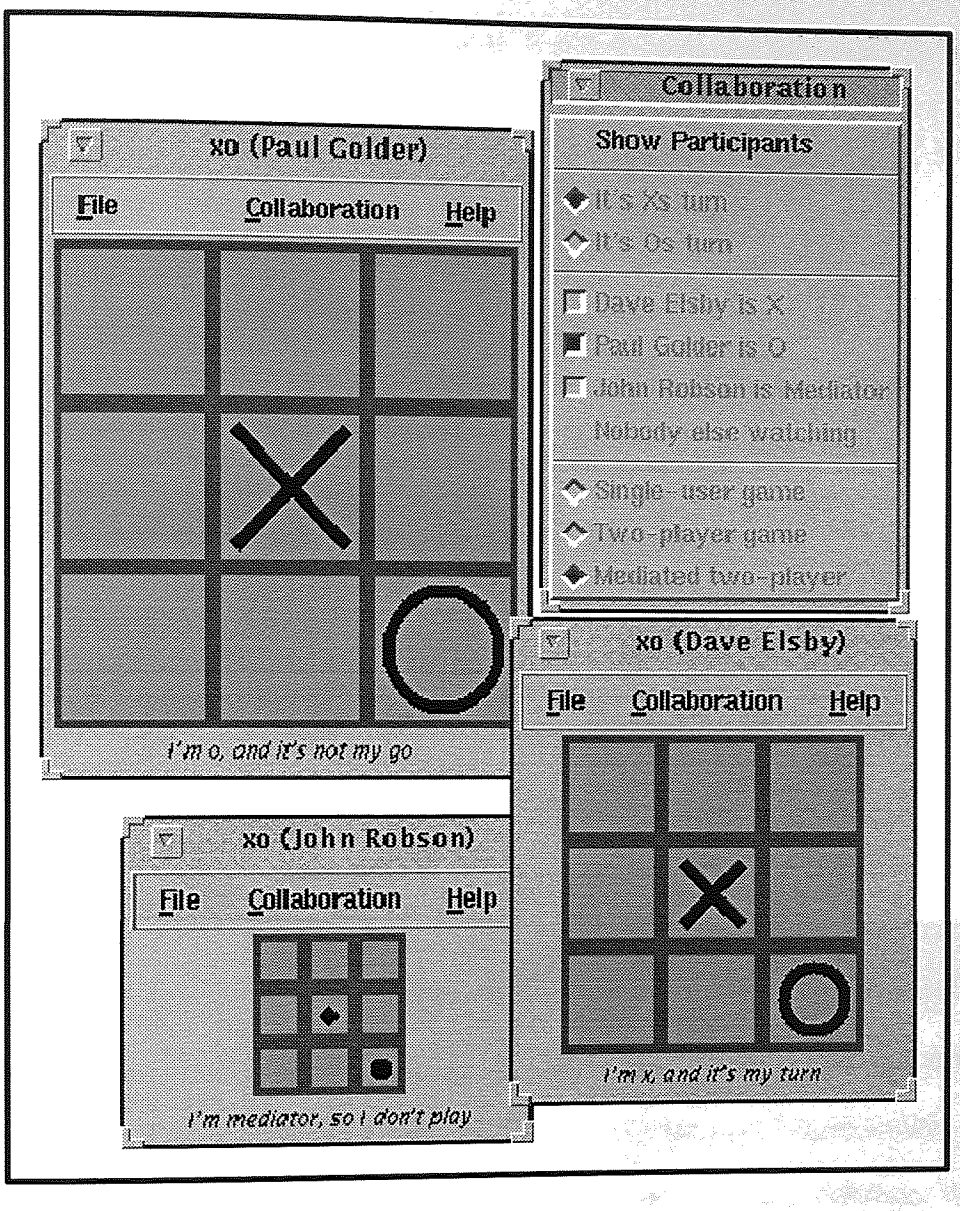

Figure 26: Tic–tac–toe game with 3 conference applications running after 2 processes have departed.

If more users leave, the world changes again. If two more users depart, for example, the remaining user will take on all the roles in the world, equivalent to the state when the first user entered the world.

5.4.2 Summary and Further Enhancements

The tic–tac–toe program presents a simple example of environmentally–aware applications, each adapting to the world they are present in. It details some of the basic capabilities of environmentally–aware software, namely:

- **Object informed about its surroundings**

  Each object knows who is in its world. What is regarded as a 'world' can vary, however. For example, each section in a document could be regarded as separate worlds. As a user traverses between sections, so their capabilities may change depending on who else is present in that section.

- **Contains a basic knowledge**

  Each object knows how to perform basic roles and actions. Objects are therefore not dependant on other users being present in order to work at a basic level.

- **Can have specialised capabilities**

  As well as a basic knowledge, certain objects can have specialised capabilities that enhance their basic functionality. For example, in an authoring environment objects may have a basic facility to mediate problems (such as electronic mail functionality). Another object could have a specialism in mediation, however, and enhance this basic functionality with voting systems and issue spaces.

- **Relinquishes roles to others as they enter the world**

  As objects enter a world, so objects present there may relinquish some of their roles and responsibilities. In the tic–tac–toe program this happens automatically, but could alternatively be achieved through invitations or requests. In an authoring environment, if two objects are in conflict, for example, they may invite a third object to mediate for them. Otherwise, an object specialising in mediation might offer to mediate in the conflict.

- **Changes roles dynamically as the world changes**

  As objects enter and leave the world, so the objects change their states dynamically. If two authors are working on a section of a document, for example, the system could automatically enter a tight–coupling mode of close collaboration. If one of the authors leaves, the system could automatically switch back to a single–user mode for document authoring.

As a further enhancement, environmentally–aware software objects could also "learn" new abilities. That is, if part of their basic knowledge is to acquire knowledge from other objects, one object could tell another object how to perform certain specialised actions. In the tic–tac–toe world, for example, the basic knowledge of the objects could be on how to learn, how to place an 'X', and how to place an 'O'. Another object entering the world could then "teach" the objects present what the rules of the game are.

5.5 The Development of a simple Collaborative Application

This section details the development of a simple collaborative application, namely the tic–tac–toe application presented in the previous section. This application is built based on the GroupKit collaborative model, using its programming abstractions and containing the rudimentary requirements for environmentally–aware software. While this section does not detail all the stages in the development of the program, it explains the use of the programming abstractions, open messaging protocols, and the use of Tcl/Tk – a simple yet powerful scripting language for rapid program development. A complete source listing for the application is given in appendix A.2[2].

5.5.1 Tic–Tac–Toe Basics

The tic–tac–toe program is a simple version of the popular 'noughts and crosses' game. The game contains a number of the basic functions required for a groupware application, as depicted in figure 27 below.

[2] The tic–tac–toe application is also available in the GroupKit software distribution under "Contributions...". Details on how to obtain this distribution are given in Appendix A.1.

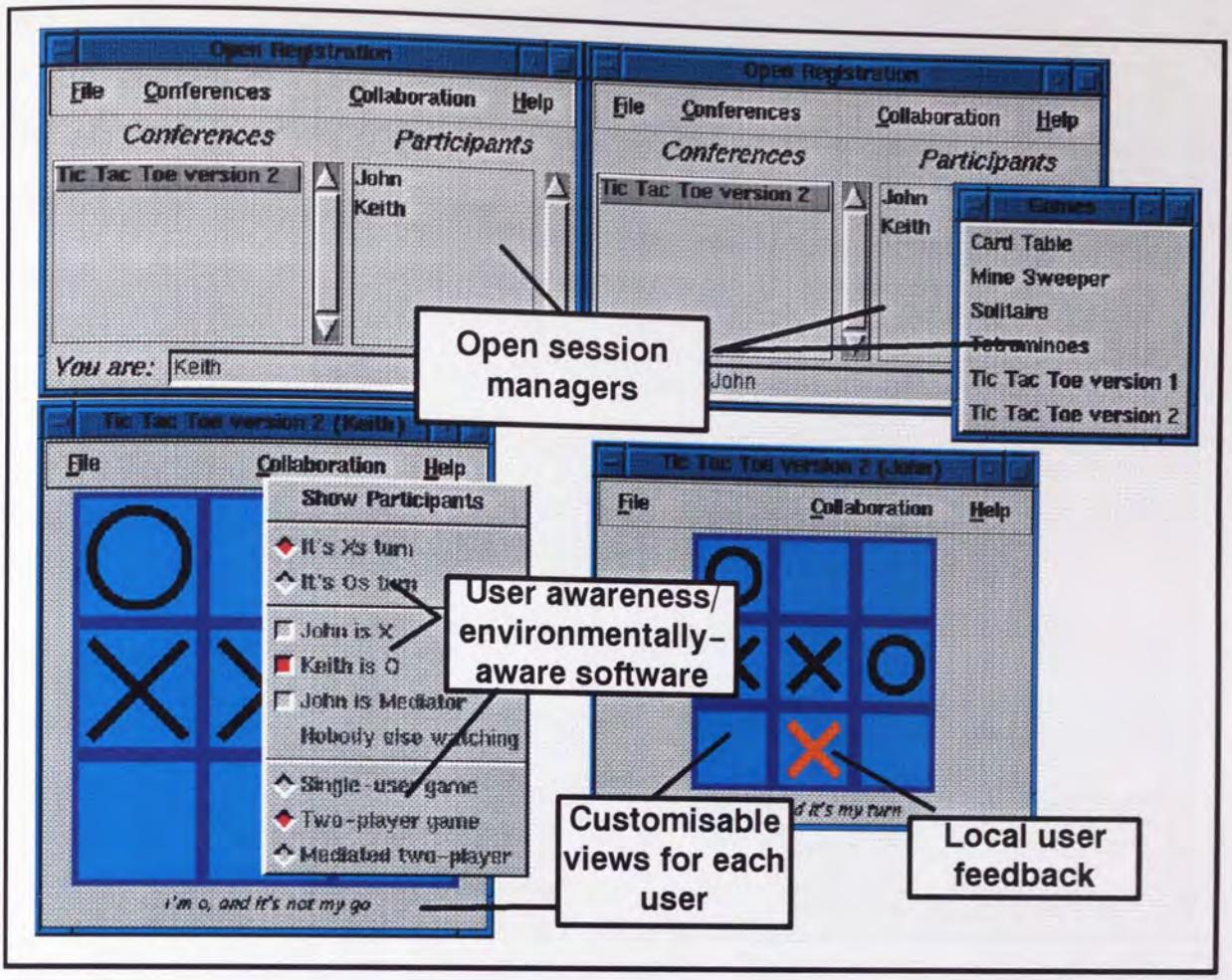

Figure 27: Basic functions required in a groupware application.

These basic functions are:

- An open collaborative session allowing users to join or leave the world as they wish.
- Group awareness (depicted in the Collaboration menu), showing who is present in the session and what their roles in the session are.
- Individual feedback, represented by highlighting where a user is moving their mouse to on the game board (if it is their turn). Each user can also set a preferred colour for themselves, which is used for this feedback. This mechanism can also be used to show users those positions where they can place a tile on the board.
- Customisable views, where users can locally change the size of the board being displayed.
- Environmental-awareness, where the application is able to adapt depending on what other users are present in the world. Examples showing the flow of the program running as participants enter and leave the world has been covered previously in section 5.4.1 above.

5.5.2 Tcl/Tk Scripting Language

Tcl (Ousterhout, 1994) is a simple yet powerful interpreted scripting language for the development of interactive applications. Tk, its companion component library, is an object–based graphics primitive/widget toolkit for the X–window environment. Together, both elements form a tool which aids in the fast prototyping and development of graphical X–window applications.

Tcl has been adapted to provide distributed processing through the Tcl–DP set of extensions (Smith et al., 1993), making it well suited for collaborative application development. The GroupKit toolkit is itself built on a mix of Tcl/Tk and C, with C being used primarily for bottle–neck areas in the system. Roseman (1993) has detailed a number of reasons why Tcl/Tk was chosen as a basis for building GroupKit on, of which the main reasons are:

- **It is quick to design new interfaces with.**

  As groupware systems are notoriously difficult to build correctly, an iterative prototyping approach is essential for building groupware applications. Groupware interfaces need to be designed quickly, and redesigned often. Also, the simplicity of Tcl/Tk (and its interpreted nature) may make it more easy for end–users to redesign or personalise their individual interfaces.

- **Tcl commands as the communications protocol.**

  Generally, a sender application needs to convert from one type (e.g. an integer) to another for communications transport, with the receiving application then converting the type back in order to use it. Tcl uses just one data type – a string – for all command processes. As a result, no conversion processes are needed and commands may be sent verbatim over sockets and executed by the receiver's interpreter.

- **Flexible binding mechanism.**

  Tcl/Tk allow developers to change the bindings of widgets in order to refine (or replace) their behaviour. This is an alternative to that of creating new widget subclasses (as in

object–oriented systems) to modify the behaviour of the objects, and is a powerful mechanism for experimenting with new behaviours for the widgets.

- **Easy separation of interface from application.**

  Groupware applications need to be able to separate the interface from the application, as different users may have different interface views onto the same data. Tcl provides a mechanism (using the 'trace' command) to generate events whenever specific variables change. Subsequently, an interface can find out automatically about changes to an underlying object.

Due to the interpreted nature of Tcl, changes in an application may be tested without rewriting and re–compiling the underlying code. In groupware testing, this is important for testing both the system and user–interface with groups of people simultaneously.

From a programming perspective, the dynamic attributes of the Tcl binding mechanism also enables widgets to take on different meanings depending on the situation without having to destroy and re–create an object representing the new class. For example, in an authoring environment a text widget may be bound so that characters are entered into it when there is just one single user. With multiple users, however, the widget binding might be changed so that the key–press events of each user are instead passed through a floor control protocol before being sent further to the widget. Such changes to the binding could occur automatically on the entering and leaving of users to a node, meaning that the behaviour of the widget would change dramatically as users enter and leave, but the widget object itself stays the same.

Such a binding mechanism allows widget sets to be developed as environmentally–aware objects. By adding an event–detection layer to the standard single–user widgets (for determining when the 'world' changes) and through the use of dynamic changes to the widget's binding, single–user widgets can be reused an adapted for use in collaborative situations.

5.5.3 Open Protocols

Open protocols are a programming implementation technique based on the client–server architecture which allows flexible collaborative applications to be developed so that they may be customised at a later date (Roseman & Greenberg, 1993).

The open protocol architecture consists of three components: a controlled object (the server) that maintains the state, a controller object (client), and a protocol describing how the two communicate. Thus the server simply executes a set of generic commands from the clients according to its protocol.

As an example, a brainstorming tool could be implemented using open protocols where the server contains a list of the ideas generated, and to which clients can direct messages using protocols such as 'read idea', 'write idea', 'update idea' and 'delete idea'. Assuming the server idea list consists of [ideaId, authorId, posInList, ideaText], then different versions of the brainstorming client could implement different types of session, none of which require the changing of the server code. Table 5 below depicts this using an anonymous client, a public client, and a personal client for two of the protocols: 'update idea' and 'delete idea'.

	update idea *(ideaId,authorId,posInList,ideaText)*	**delete idea** *(theAuthor,ideaId)*
Anonymous	**Display**(ideaId, authorId, posInList, ideaText)	*do nothing*
Public	**Display**(ideaId, authorId, posInList, ideaText)	*do nothing*

Personal	**if** (authorId=MY_ID)
	  set textcolour red
	else
	  set textcolour black
	Display(ideaId, posInList, ideaText)
		if (theAuthor=MY_ID)
		  DeleteIdea(ideaId)

Table 5: Code for three different brainstorming clients showing the behaviour they display on receipt of an 'update idea' protocol and 'delete idea' protocol.

Adapted from: M. Roseman & S. Greenberg (1993), *Building Flexible Groupware through open protocols. p.5*

In the tic–tac–toe application, open protocols are used to determine who's turn it is, whether a user can place a tile, and evaluate user requests to place a tile. Instead of having a central server

with clients linking to it, each tic–tac–toe program contains both client and server code in its core, although it will always direct its protocol requests to the game mediator. The game mediator can, however, change during a session as users enter and leave the world. A client could thus direct a message to itself (as in a single–user game) or to a client acting as mediator (as in a mediated game).

5.5.4 Programming Abstractions and Data Structures

The tic–tac–toe application uses two environment variables. One defines the game 'world' globally while the other defines the views onto the world that each participant has. Both environments are 'bound' so that events are generated in the application whenever their state changes. Table 6 below details these two environments, and describes the data structures that each contains.

Environment	State	Data Structures	Description
envXOWorld	shared	gametype	The type of game in progress – either single–user, two–player, or mediated
		turn	Whose go it is – either X, O, or game-over
		players.*t*	The application numbers of the game participants for player type *t* (X, O, mediator, list of watchers)
		board.*x.y*	The game board, consisting of a 3x3 *x,y* grid holding blank, X, or O values
		gameover.winner	Who won the game (X, O, or draw)
		deletedUser	Special flag for when a user leaves so that only one of the remaining applications *initially* deals with the event
envXOView	individual	boardSize	The size of the game board
		lastX, lastY	The last x,y coordinate the mouse was over on the game board (for the addition and removal of highlighting effects)

Table 6: Basic environment data structures in tic–tac–toe.

By making the envXOWorld environment shared, changes made to any element in the environment are automatically propagated to the other applications running in the session.

This means that only one application needs to deal with a particular event (such as a person entering or leaving the world). Changes made to that application's environment structure after dealing with the event are therefore automatically sent to the other applications.

By making the envXOWorld environment individual, each application in the session can have a different view of the world without affecting other applications in the session. In this tic–tac–toe program, different views are represented by the ability for each person to have different board sizes in the session. Such a separation of the view from the underlying data structure would also enable other types of board, such as a text only view of the game on a terminal, to be represented.

5.6 Conclusions

This chapter has focused on the implementation issues for collaborative applications. It has detailed the basic collaborative architectures and their related issues and described a groupware toolkit to aid in the development of collaborative applications. Furthermore, a model for an "environmentally–aware" software application was presented, along with an example of its use. Finally, the main elements of a simple collaborative application were noted, with details of how the program was developed and the main data structures used by it described.

The following chapter brings together the theories presented in this chapter and chapter four, and taking into consideration the literature from chapters two and three, to detail a collaborative application developed to aid people working in a multi–user authoring environment.

Chapter 6

Collaborwriter

6.1 Introduction

Collaborwriter is a collaborative authoring system which has been developed in response to many of the issues presented in chapters four and five, and with regard to the group and writing processes identified in the literature of chapters two and three.

Collaborwriter does not consist of just one application program, but instead links together a number of programs to form an authoring environment. This chapter is divided into sections representing the individual program elements, and how they interlink to form the whole system. Thus, the chapter first presents the process/network architecture used by the system, followed by some technical issues regarding transaction management, locking and groupware widgets. The top–level registration and project–management policies used in the system are then discussed, followed by the authoring tools and such issues as version control. Finally, the negotiation and consolidation tools are described, detailing how they are used to aid conflict resolutions which may have arisen during the authoring process.

6.2 Top–level System Architecture

Collaborwriter is built based on the GroupKit groupware infrastructure described in chapter five, but with a number of extensions. The system thus consists of:

- **A central registrar**

  The registrar runs on one 'well–known' machine, and which other users connect to. As well as containing information about the session managers connected to it, this registrar has been extended to act as a central server for the high–level data stored by Collaborwriter, and which is used by the project management session managers. By using

a central process for storing the initial information required by Collaborwriter, consistency across all the clients connecting to it is ensured.

- **Project management session managers**

  The session managers form the core of the project management policies used by Collaborwriter, managing the control of the conference processes in the environment. They determine who can connect to or join other conference processes, who can remove conference processes, and how these processes are managed. The session managers connect to the central registrar and retrieve the information contained in its server, resulting in the information being replicated across all the clients. If any changes are made by a session manager, the change is first sent to the registrar and then distributed to all the connected clients. Provisions need to be made, therefore, for the receipt *at any time* of new information by clients currently connected, and for those instances where users are not currently connected to the registrar.

- **Various conference processes**

  Conference processes form the productivity core of the environment. These processes can be based on any conferencing tool created using the GroupKit environment, such as an authoring tool, a shared drawing tool, or a negotiation tool. Conference processes can talk to one another under the control of the session manager, so that the same tools can be used simultaneously by different groups without conflicting results.

The control flows between three users and the three process types for each user are depicted in figure 28 below.

In this figure, there are three projects: X, Y and Z, with only one of the projects (project Y) being shared by the three users. The session managers know what projects are available from the central collaborwriter data held in the registrar, and can thus manage the conference processes from this information. Subsequently only the authoring tool used in project Y is shared between Paul and Barbara (although Keith could also share it) yet only Paul has the

rights to use the authoring tool from project X. The negotiation tool is shared between all three users of project Y, with the graphics tool only available for use by user Keith in project Z.

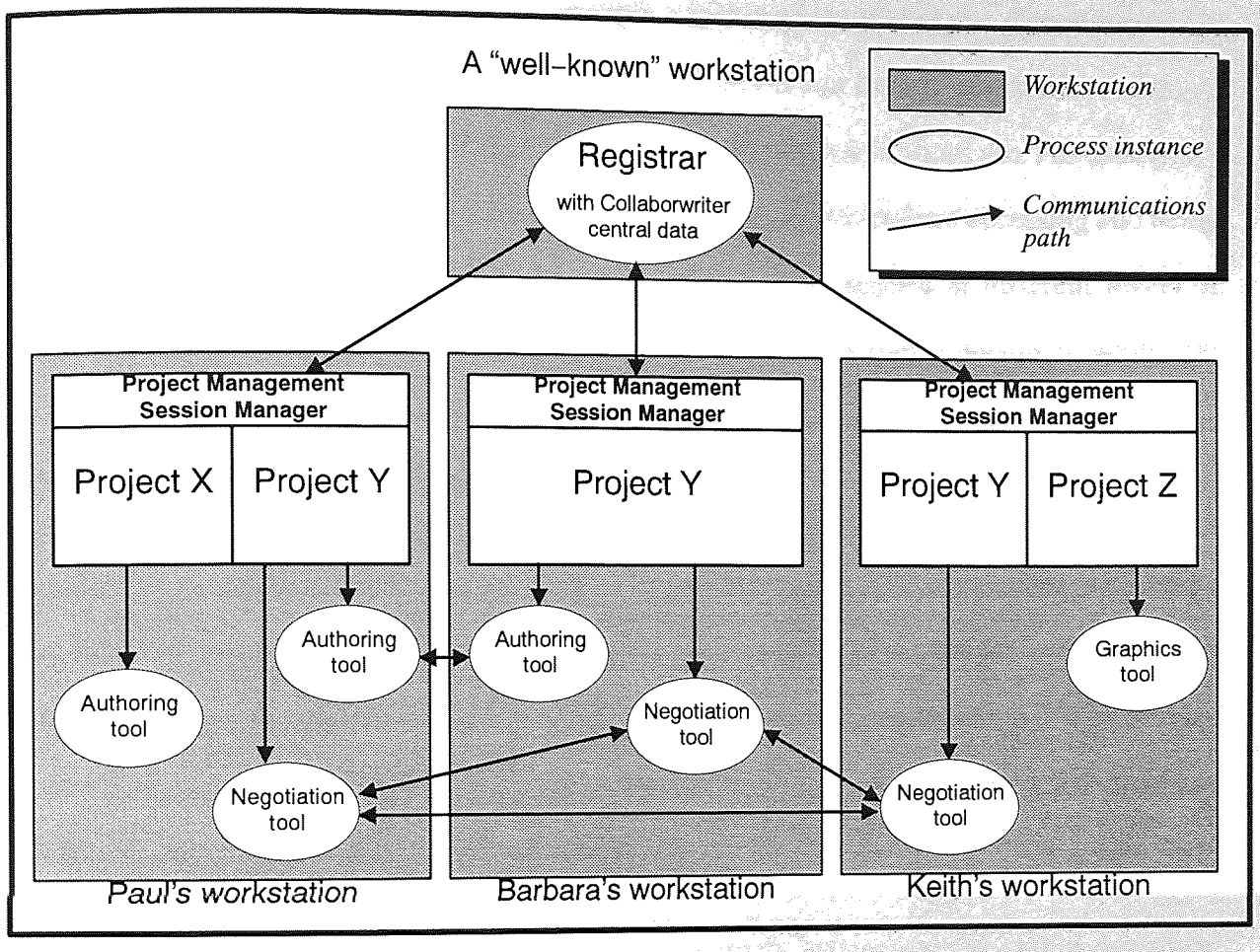

Figure 28: Collaborwriter's runtime process model between three users working in three separate projects.

6.3 Groupware Environments and Widgets

6.3.1 Environments

The environment structure as used by GroupKit has been described in chapter 5. GroupKit provides a mechanism, using models found in computational reflection (Kiczales, 1992), through which environment commands may be modified and extended (Roseman, 1995). This model has proved useful in extending GroupKit environments for use in a number of cases.

6.3.1.1 Environment Locking and 'Race' Conditions

Any multi–user application has a requirement to perform locks on objects or data at some point. In a multi–user environment, for example, a lock might be placed on a document to stop it being accessed by multiple users. Locking mechanisms can be very restrictive, however, especially in an environment mixing both synchronous and asynchronous use. For example, a user may lock a document for hours, days or more, stopping others from accessing and being able to continue their work. To combat this, locks can be applied at different levels of granularity, and as either read or write locks. Thus a user may lock a document while still enabling others to read it, or only lock a section or paragraph in the document allowing others to read or write to the other portions.

GroupKit provides no locking mechanism and, as its main data structure is managed through the environment command set, this command set had to be extended for adding locking functionality. Due to the nature of GroupKit environment commands, a basic locking mechanism can be created by adding `lock` and `unlock` commands to the environment and by overriding and adapting the `set` command.

A problem can arise using such a mechanism, however, concerning the call to determine whether a lock currently exists. If a lock does not exist and two (or more) users *simultaneously* request a lock on an item, they will both be informed that no lock currently exists and subsequently both perform the lock. A 'race' condition thus occurs, where only one of the users' locks is actually generated. There are three basic solutions to the problem:

1. Request and grant locks through a central process;

2. After a short time delay, see if you *really* got the lock; or

3. Use a message request/receive system where you send a message event requesting a lock and at some point receive a message event granting the lock.

Each solution, however, itself raises other problems. Solution ① works well if there are fast links to the central process resulting in only small process delays. A central process must also be 'known' to the other processes and be able to handle varying degrees of 'traffic' for the

request and relinquishing of locks. Solution ② actually enforces a time lag, which may have usability consequences for the operation being performed and the success of which is determined by the speeds on the network and processor nodes. Finally, solution ③ again has a time lag but also increases the coding complexity by requiring routines needing locks to be only written through event callback handlers. This means, however, that locking events can be queued and handled in a serialized manner.

In the Collaborwriter environment, the locking mechanism used is based on the first method. The session managers have been written so that locking is not required and subsequently a conference process can be used as a 'well-known' process. Such a scheme is also used by GroupKit for serialising environment changes. Figure 29 below consists of a code fragment depicting code for implementing the locking mechanism, designed in a manner to allow locking on portions of an environment. The portion locked consists of the node requested and any subordinate nodes it contains.

```
#-------------------------------------------------------------------
# On a attempt to perform a 'set' operation,first check if environment
# is locked before doing the command. If locked, return '0' otherwise
# return the locking user's name
#-------------------------------------------------------------------

proc cw_SetEnvironment { env cmd node data } {
      set locks [$env keys locks]
      foreach lock $locks {
            if { [string first $lock $node] == 0 } {
                  #
                  # we've found a lock, so see who has locked it
                  set lockedBy [$env get $lock]
                  if { $lockedBy != "[userprefs name]" } {
                        puts "$env $node locked by $lockedBy"
                        return 0
                  }
                  break
            }
      }

      $env _cwset $node $data
}
```

```
#-------------------------------------------------------------------------
# Attempt to lock the environment, returning the name of who has/gets
# the lock.If a.b.c is locked,trying to lock a.b.c.d is disallowed,as
# is trying to lock a/a.b. Locking a.c is allowed,as is locking a.b.d
#-------------------------------------------------------------------------
proc cw_LockEnvironment { env cmd lockItem userID } {
   # if we are the unique user ID, deal with the lock request,
   # otherwise pass it on to the unique user
   if { [_gk_getUniqueUser] == [users local.usernum] } {
      set locks [$env keys locks]
      foreach lock $locks {
         if { (![string first $lock $lockItem]) ||
         (![string first $lockItem $lock])} {
            set lockedBy [$env get $lock]
            puts "$env is already locked by $lockedBy"
            return $lockedBy
         }
      }
      $env set locks.$lockItem $userID
      return $userID
   } else {
      return gk_toUserNum [_gk_getUniqueUser]
                              $env $cmd $lockItem $userID
   }
}
```

```
#-------------------------------------------------------------------------
# Unlock the environment, checking that we had the original lock
#-------------------------------------------------------------------------

proc cw_UnlockEnvironment { env cmd lockItem userID } {

   # if we are the unique user ID, deal with the lock request,
   # otherwise pass it on to the unique user

   if { [_gk_getUniqueUser] == [users local.usernum] } {
      set lockItem [lsearch [$env keys locks] $lockItem]
      if { $lockItem != -1 } {
         set lockedBy [$env get locks.$lockItem]
         if { $lockedBy == $userID } {
            $env delete locks.$lockItem
            return $userID
         } else {
            puts "$env not locked by you, but by $lockedBy"
            return $lockedBy
         }
      }
      return ""
   } else {
     return gk_toUserNum [_gk_getUniqueUser]
                              $env $cmd $lockItem $userID]

   }
}
```

Figure 29: Fine-grained environment locking through a central process.

6.3.1.2 Environment Transactions

Transaction management concerns the grouping of sets of related commands into a '*packet*', with the packet either being fully handled or not handled at all by the system. It is another area of importance in collaborative systems which has been partly addressed in database theory.

Database systems use transactions for ensuring database integrity, such as in a financial transaction for the completion of both credit and debit transfers between two bank accounts. Collaborative systems can use transactions for both integrity reasons and communication reasons. For example in GroupKit, as soon as an environment changes the change is passed throughout all running processes. In certain cases this change notification may not be desired, however, with coding 'work–arounds' being used instead. Two cases of such work–arounds are used in the Collaborwriter session manager code for:

- creating new icons; and
- creating a voting issue.

In the first case, when creating a new person, team or project icon, the `collaborwriter` environment structure has an `<item type>.<item name>` entry added to it which includes multiple options. As each option is set, an event is triggered to update the environment which causes the icons to be created on all the other desktops. The icon must first exist before its attributes are changed and so there is a `'created'` field in its attributes list. This field is set first in order to ensure that the icon gets built, with all other icon initialisation requiring the icon to exist being performed in a specific order so that the icon is initialised as required.

With a transaction method, all option settings could be grouped together and then 'fired' for notification throughout the system. This removes the need for careful ordering of the items and that, instead of multiple communication broadcasts, only one communications event is sent between all the processes for the whole of the transaction.

When creating a voting issue a similar state occurs where multiple items need to be set before triggering the 'environment change' event. The method used by the voting tool is to contain

all the information for a voting issue in a single list and then communicate that list. The disadvantage with using this method is that list traversal is required for extracting the information and that 'magic numbers' or global constants are needed to identify what each item at list index '*x*' means.

Figure 30 below shows a code fragment for an alternative strategy to environment transaction management in GroupKit. The code is implemented as a further extension to the GroupKit environment and consists of two additional environment commands: `begintransaction` (where the transaction commences) and `sendtransaction` (where the transaction is finished and the changes transmitted) along with extensions to the GroupKit `set` command.

```
# set up our notifier object and bind it so we receive details of any
# transaction sends
gk_notifier environmentTransaction
environmentTransaction bind transaction
                        "cw_ReceiveTransaction %E %N %P"
#-------------------------------------------------------------------
# Initialise the new environment commands and override the set command
#-------------------------------------------------------------------
proc cw_InitialiseTransaction { envName } {
   $envName command set begintransaction "cw_BeginTransaction"
   $envName command set sendtransaction "cw_SendTransaction"
   $envName command rename set _cwtransset
   $envName command set set "cw_SetTransaction"
}
#-------------------------------------------------------------------
# If in transaction,add the change to a packet of data for later send
#-------------------------------------------------------------------
proc cw_SetTransaction { env cmd node data } {
      if { [$env option get transaction.name] == "" } {
            $env _cwtransset $node $data
      } else {
            set packet [$env option get transaction.packet]
            lappend packet [list $node $data]
            $env option set transaction.packet $packet
      }
}
#-------------------------------------------------------------------
# Initialise the transaction data structure
#-------------------------------------------------------------------
proc cw_BeginTransaction { env cmd transaction } {
      if { [$env option get transaction.name] != "" } {
            puts "Transaction is already set for environment $env"
            return ""
      } else {
            $env option set transaction.name $transaction
            return $transaction
      }
}
```

```
#-------------------------------------------------------------------
# Fire the transaction- use notifier to send to all linked processes
#-------------------------------------------------------------------
proc cw_SendTransaction { env cmd } {
   if { [$env option get transaction.name] == "" } {
      puts "No transaction to fire for environment $env"
      return ""
   } else {
      environmentTransaction notify transaction [list [list E $env]   \
            [list N [$env option get transaction.name]]                \
            [list P [$env option get transaction.packet]]]
      $env option delete transaction
   }
}

#-------------------------------------------------------------------
# Receive notification of the transaction. Set a flag so that further
# notifications aren't sent and set the transactionName item to
# trigger any other linked events
#-------------------------------------------------------------------

proc cw_ReceiveTransaction { env transactionName packet } {
      set origState [$env option get inhibit_notify]
      $env option set inhibit_notify yes
      foreach p $packet {
            $env set [lindex $p 0] [lindex $p 1]
      }
      $env option set inhibit_notify $origState
      $env set $transactionName [$env get $transactionName]
      puts "Transaction $transactionName sent to $env"
}
```

Figure 30: Environment transaction routines

The code relies on the fact that the option stream in GroupKit environments is not shared between users and can therefore be used as a private data store. When a transaction is created, instead of the `set` command writing to the environment, the changes are written to a list held in this private store. Then, when the transaction is sent, a GroupKit 'notifier' object is set to contain the details. Each process initially sets a binding on this object which is then triggered on the change to call the `ReceiveTransaction` procedure. This procedure in turn uses the inhibit_notify variable in the command to ensure that only this process gets the changes made to its environment and does not ask for notifications to be sent to all the other processes.

6.3.2 Widgets

Collaborwriter adds some additional user–interface widgets to the suite of widgets available under GroupKit. These widgets are built to be 'environmentally–aware' in that they are self–contained and will act and react differently depending on how many users are accessing the same widget, changing as users arrive and leave from the node.

The additional widgets are the GroupIcon widget, the NodeIcon widget, the GroupTable widget, the PictureBar widget, the IbisIcon widget and the Multitext widget.

6.3.2.1 The GroupIcon Widget

The GroupIcon widget is a composite widget consisting of an icon (or some other widget type) with some text underneath. It is the core environmentally–aware widget, and incorporates an internal GroupKit environment structure, routines for loose and tight coupling with the widget, and binding, state setting and retrieval mechanisms for all its internal widget components.

Mouse button 3 is bound to the widget for motion events, enabling the widget to be dragged around its parent canvas. The loose and tight coupling of the widget determines whether the widget position is updated on other users' displays when dragged, or whether each users' displays have independent positioning of the widget.

6.3.2.2 The NodeIcon Widget

The NodeIcon widget inherits all the features of the GroupIcon widget and incorporates features and routines for building one–to–many tree relationships. Thus the widget includes linked lines between nodes, maintains parent–child lists and adds commands for tree highlighting, member detection and redrawing.

Mouse button 3 is bound to the widget for double–click events which causes the tree from that node down to be tidied and redrawn in an intelligent manner.

6.3.2.3 The GroupTable Widget

The GroupTable widget is a widget designed to improve overall author awareness and is incorporated with the NodeIcon widget explained above. The widget is depicted in figure 31 below, where three authors are working in node "Section 1.1" while no–one is working in node "Section 1".

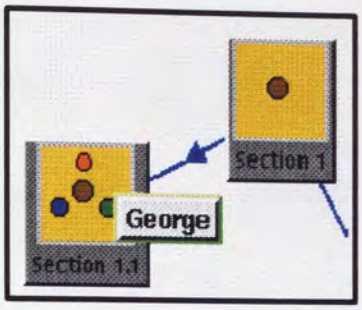

Figure 31: A grouptable widget in part of a node tree hierarchy

This widget represents a conference table with various participants sitting around it. As people enter and leave the 'room' so the widget changes to depict the current members. Each user is represented using their preferred user colour, with popup menus over the individual members listing who that member is or, if called over the central table, showing all the members in the room.

Figure 32 below shows how the GroupTable widget changes as three users enter a node, one after another.

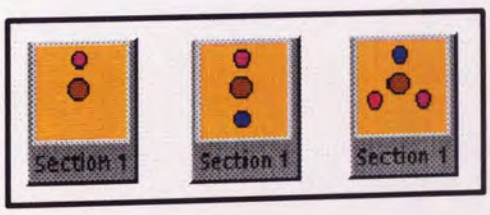

Figure 32: The sequence of events as three users enter a node containing a GroupTable widget

By using this widget authors can quickly see an overview of which nodes other people are interested or working in without having to enter each respective node.

6.3.2.4 The PictureBar Widget

The PictureBar widget is similar to the GroupTable widget in that it is designed as an aid to author awareness, depicting the same information as the GroupTable widget but in a different format. The PictureBar widget shows graphically who is in a particular node, using pictures chosen by each user to represent themselves. Figure 33 below shows a typical example of the PictureBar widget where three users are in the same node.

Figure 33: PictureBar widget representing three users in a node.

As people enter and leave the node, so the PictureBar automatically changes to show the latest users. Clicking on a picture also details a person's name and shows what their colour preference is (which is used by the other widgets for depicting that person). Groups of pictures on the picturebar may be selected together, and a hook is provided where a script may be executed on double–clicking over one of the pictures. Such a script could be broadcasting an electronic mail message to the selected users or requesting the selected users to perform a particular action. Thus while the GroupTable widget gives an overview of who is in a node using a small display overhead, the PictureBar widget gives a detailed overview and allows the manipulation of who is present in a particular node.

6.3.2.5 The IbisIcon Widget

The IbisIcon widget is designed around the *Issue Based Information System* (Rittel and Kunz, 1970) methodology. The widget is used by the negotiation tool as an aid to mapping issues, positions and arguments between users and improve the visibility of the decision–making process.

The widget consists of an introductory subject text box along with a number of buttons underneath. The widget frame and subject text changes depending on the containing entity

type (either an issue, position or argument). The buttons underneath represent a person's view on the item. They are based on an extension to the checkbutton widget (called a "choicebutton") and allow users to either agree, disagree or remain undecided about a particular topic. Figure 34 below shows examples of the widget entity types.

Figure 34: The IbisIcon widget entity types.

In a similar manner to the GroupTable widget, as users 'see' the IbisIcon widget, a choicebutton with their user colour is added to their and all other users' IbisIcon widgets. On exiting from the widget their decision persists, allowing the negotiation and decision–making process to be made non–simultaneously.

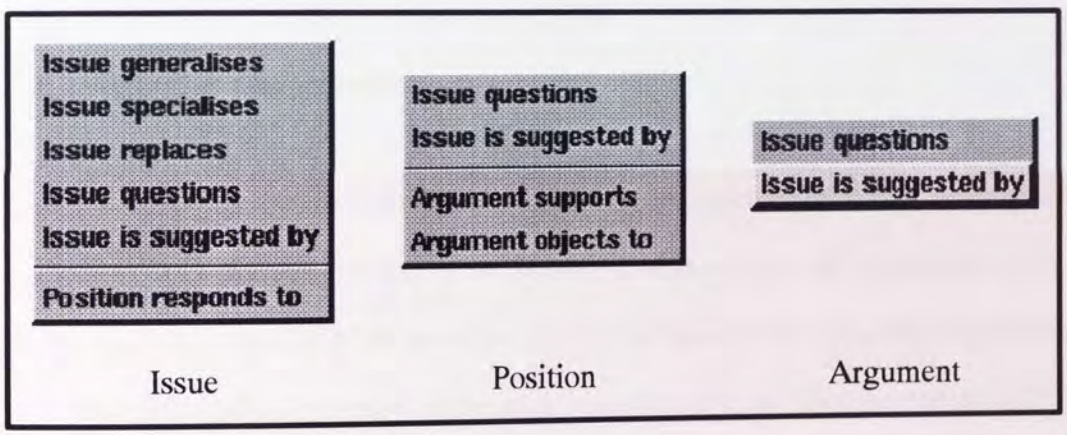

Figure 35: The popup menu types for each IbisIcon entity.

An IbisIcon widget may also be linked with another widget in a similar manner to NodeIcon widgets. Only certain entity types may be linked to one another, however, so an argument entity can not be directly linked to another argument entity. As an aid to the user, an IbisIcon can create a new entity and subsequently link the new entity to itself through a set of entity–dependent popup menus on the widget. The menu choices are depicted in figure 35 above.

The links created by the widget are named links, with either link text or an icon appearing between the two entities. For further information on the widget, a table showing the link types

and their icon representations is presented in the discussion on the negotiation tool, (section 6.6.1, tables 8 – 10). This section further describes the decision–making options available to users via the widget and accessible through the negotiation tool.

6.3.2.6 The Multitext Widget

The Multitext widget inherits its properties from the GroupIcon widget and is based on a version of the GroupKit gkText widget. The widget is central to the authoring tool and controls the locking and revision management between users in a simple freeform text box. It also controls the mouse and keyboard bindings in the widget for keyboard shortcuts and mouse operations on the text. Furthermore, author awareness is established by highlighting which line each user's insertion point is at inside the widget body.

6.3.2.6.a Revision Management

The revision management element is based on editing sessions, where a new editing session is created whenever a node is entered. Thus, if a user enters a node and types some text, this is entered in an editing session. If the user changes the text just typed, these changes are regarded as part of the same editing session. If the user exits the node and then re–enters it, however, making the same changes would now be regarded as making the changes in a new editing session (with the ability to roll back to the previous version).

In a multi–user environment, however, a new session will be automatically created if a change is made that intersects with a change made by another user. Figure 36 below depicts this graphically.

In the first case, author A changes the word 'initial' to 'starting'. He then changes his mind, changing the word 'starting' to 'start of the'. A new editing session is not created here and, if the version was rolled back, it would revert to `"This is the initial text."`.

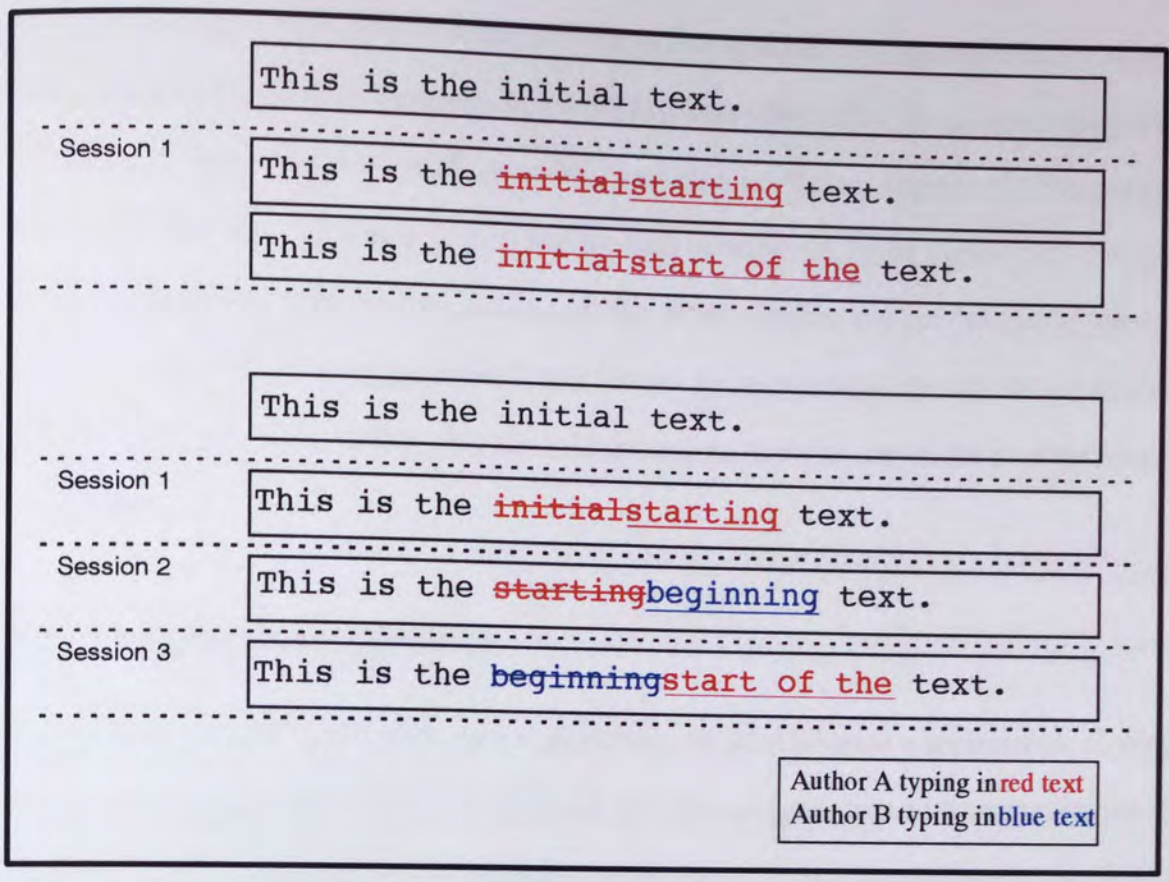

Figure 36: Revision management sessions for one user or two users on a portion of text.

In the second case, author A again changes the word 'initial' to 'starting', causing a new session to be created. Author B sees this change and changes the word 'starting' to 'beginning' which causes another new session to be created. Author A decides to change the text again, choosing to use 'start of the' and causing a session to be again created. If the changes were to all be rolled back, the following would result:

```
"This is the start of the text."=>
"This is the beginning text." =>
"This is the starting text."  =>
"This is the initial text."
```

Visually, only the changes of the last session are shown, with additions shown as underlined text and deletions shown as strike-out text using the author's user colour. This system could be extended to provide a dialog of hierarchical changes.

6.3.2.6.b Text locking

The revision management method described above can add unwanted overhead to the system, with authors wishing that other authors are unable to change what they are currently working

on. To reduce this, the Multitext widget provides a fine–grained locking mechanism where an author can make a selection and request it be locked from other edits. By using the selection as the locking range, an author can choose to lock just a few words or paragraphs, or the complete text in a node. Other users may request that the lock be removed, or the system can relinquish the lock after a user–specified period of inactivity. When a lock is in place, the text appears in a shaded variant of the locking author's user colour so as to easily identify the person who initiated the lock. Any number of separate locks may be made by any number of authors in a single node.

6.4 Project Management

Project management is one of the key requirements for a collaborative application. A project management policy must be flexible enough to be both open and strict with what users are able to perform. For example, users may wish for one project to be totally open, where any user can join and perform actions in the project without the express permission of another user. Alternatively, it may be preferable to have closed, tightly–controlled projects, with limited user access.

Collaborwriter builds its project management policies through the use of three basic entities:

- **People**

  People are the basic element of any collaborative endeavour. Collaborwriter allows, through a simple drag–and–drop interface, people to be added to either teams or projects, and modified through dialogs to have a number of default roles. Collaborwriter also maintains a special anonymous user, called "Anybody" which allows users to provide default actions whenever a new person enters the collaborwriter session.

- **Teams**

  Teams are collections of people. A team consists of at least one team leader (a person who controls the operations available internally to a team), other people as team members, and any number of other teams. Teams are not hierarchical structures, however, but are more consistent with set theory.

- **Projects**

  Projects are entities in which people or teams collaborate. A project is a hierarchical structure, and consists of at least one project leader (who controls the operations available internally to a project), people and teams as project members, sub–projects, and what conference tools are available to that project. This last attribute is similar to a Rooms (Henderson and Card, 1986) interface, where different projects can take on very different 'flavours' depending on what tools are available.

6.4.1 Session Manager User Interface

The project management session managers control the creation and interaction of these basic entities using an iconic desktop metaphor. Figure 37 below shows the interface presented by Collaborwriter for one user. This consists of three independent windows: Projects, Teams and People. People icons are added automatically as other users start Collaborwriter sessions, and persist once they close their sessions.

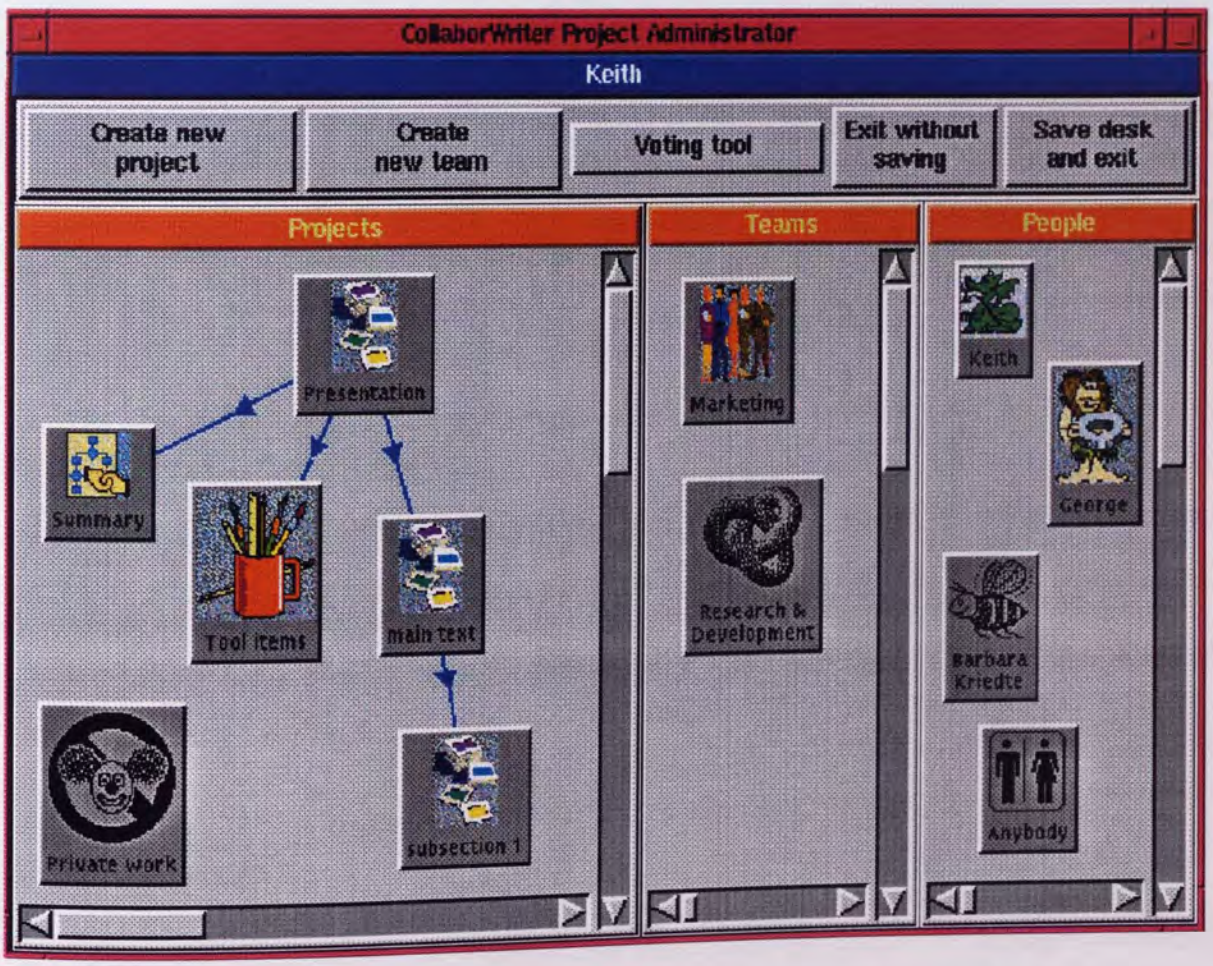

Figure 37: The Collaborwriter project management session manager interface.

As a result, a person must first open up a Collaborwriter session before being able to be manipulated by other users (such as added to a team or project). Thus, when viewing the display in figure 37 above for user Keith, it shows that users George and Barbara have previously logged in (or are currently logged in) to the system.

Teams and projects are created by clicking on the appropriate button, and then modified by dragging–and–dropping the required entity onto the icon. For example, the 'Research & Development' team, consisting of George and Keith, can be easily created by creating the team and then dragging the George people icon and dropping it over the team icon.

All icons on the desktop can be manipulated in the same manner, with the mouse commands available to each icon detailed in table 7 below. Some commands however, such as the Properties dialog, will differ depending on the icon clicked on when instantiated.

Mouse Operation		Resulting Operation		
		People	Team	Project
Left button	Single–click	–	–	Open specific tool
	Drag	Drag the icon for dropping onto another icon		
	Double–click	–	–	Open authoring tool
Middle button	Single–click	–	–	–
	Drag	–	View people in team	View people in project
	Double–click	Open appropriate Properties... dialog for the icon		
Right button	Single–click	–	–	–
	Drag	Move the icon around in the canvas window		
	Double–click	Tidy up any sub–icons attached under this one		

Table 7: Mouse button operations available on an icon

The desktop used by each user can be individually customised, with the placement and pictorial representation of icons, and saved for future sessions. Thus, whilst an icon created by a user is reproduced on other users' desktops, the final positioning and appearance of the icon may be localised for each user.

6.4.1.1 People

People in the Collaborwriter world are represented by individual icons. In the figure above, as well as the three users' icons, there is also an icon for 'Anybody'. This icon is used to represent

anonymous users, and is used as a means of dictating the *openness* of the system. For example, the creators of the presentation may wish to enable any user to read their work. They can thus drag the 'Anybody' icon over the 'Presentation' icon and set its attributes to have read access to the presentation. Any user joining the session, even if not previously present in the Collaborwriter world, now has the ability to read the presentation.

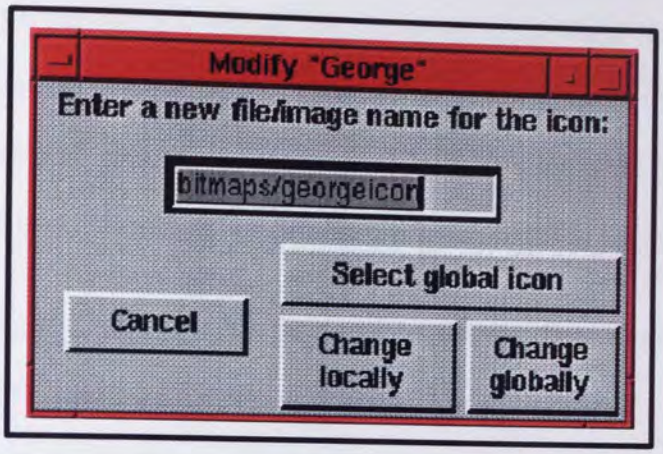

Figure 38: Dialog for changing an icon's pictorial representation.

In all cases, users can change the pictorial representation of an icon on their personal desktop by selecting its Properties... dialog (as shown in figure 38 above). For people icons, only the owner of the icon (i.e. the person it represents) is able to apply an icon change globally across all users' desktops, such as by updating their picture. In cases where an individual has already created a localised version of the icon, they will first be asked whether they wish to change to the global version (figure 39 below). They can also select the global version of the icon at any time through the Properties dialog.

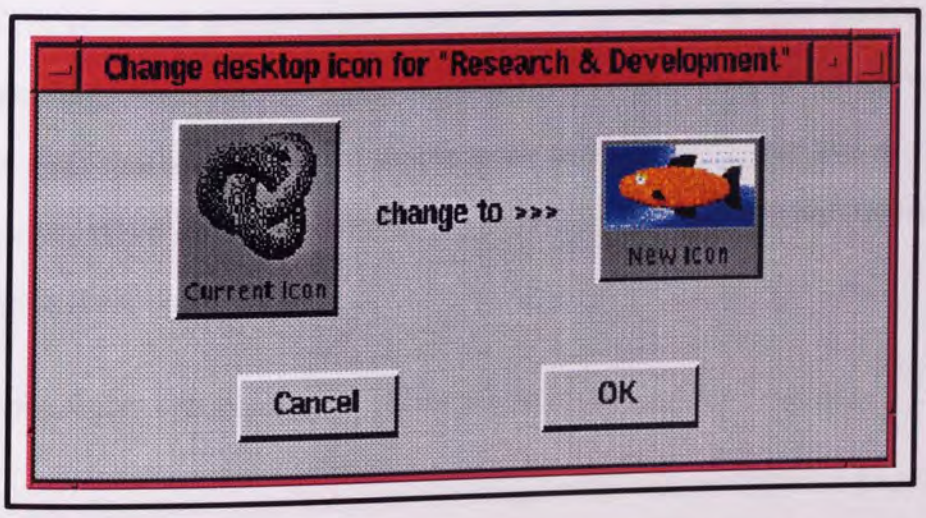

Figure 39: Dialog whether to change the current custom icon into a new global icon.

6.4.1.2 Teams

Teams consist of collections of people (or other teams), and are key to the project management concepts in Collaborwriter, aiding the longevity and fluid nature of writing projects. In a project spanning a considerable period of time, such as in the writing of European or international standards, individuals may join and leave the writing stages. Instead of finding all occurrences of the person in a system and changing them when one person leaves and another joins, only those teams which the person was part of need be changed. For security reasons this may also be important in large teams, where the members want to ensure that a particular user is removed completely from the system when they leave.

As mentioned previously, teams are not formed from hierarchical structures, but instead are based on set theory. This enables a team to consist of a wide number of people (or teams) and for people to be members of multiple teams, the same as in matrix and project structures found in many business environments. Circular references are also permitted, where team A contains team B, and team B contains team A, which would not be possible using a hierarchical model.

People in a team may either be team leaders or team members. This split enables teams to decide internally the flexibility of their membership – who can join the team, who can leave the team, and what the people may do in the team. By including the 'Anybody' anonymous user, people joining the team automatically inherit the attributes of that user – either as a leader or member. Thus, a group of people using democratic methods might consider all members as team leaders, so that all members have equal rights and anyone who joins the team enters it at an equal position. Alternatively, it may be preferable to restrict the decisions of who can join or leave a team to a select group of people in the team, in which case they are made team leaders and are asked whether a specific person may join the team. The basic policies used by the system in these cases are:

- any team leader can add a person to the team;
- if a person requests to join a team, any team leader can decide whether to give them access – if different leaders give different responses, the person is allowed to join the team;

- any team leader can remove a team member from the team;
- any team leader can 'upgrade' a team member to team leader status; and
- a team leader may request to either 'downgrade' or remove another team leader from the team. In this case, a voting issue is raised, with all team leaders being able to vote and the issue being closed on a majority decision.

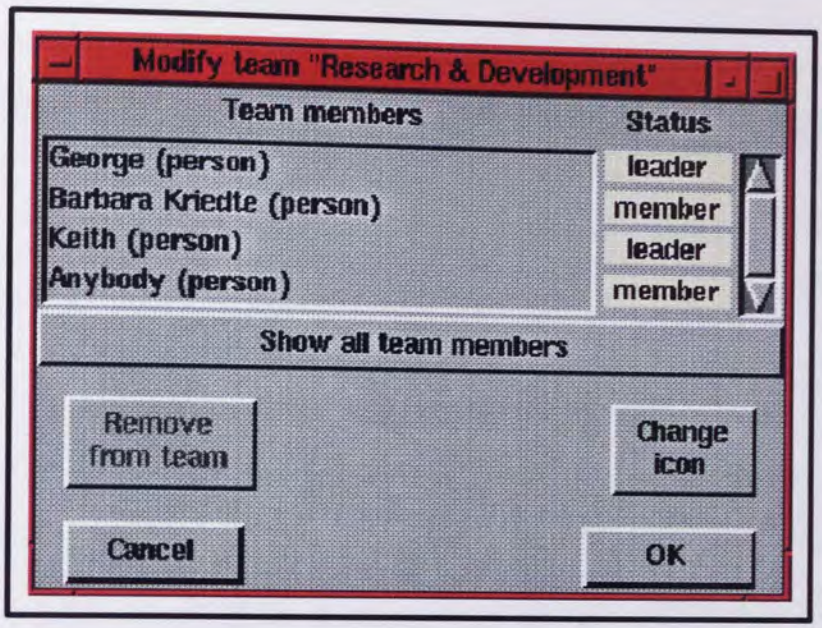

Figure 40: Properties... dialog for a team, where the person opening it is a team leader (thus the status buttons are enabled)

People are added to teams by simply dragging their icon onto the team icon, and their attributes modified by using the Properties... dialog (see figure 40 above) for the team, where the team icon may also be changed globally by any member of the team.

6.4.1.3 Projects

Projects form the working core of the Collaborwriter authoring environment. They consist of hierarchical structures containing a link to the authoring layer of the system, plus any number of entities representing people, teams, project tools or other sub–projects. By stating who can join a project, projects can be either private to an individual, to a group of people, or public to all with varying degrees of access–rights. These access levels can be changed at any time, enabling for example private projects to become publicly available near completion for comment and review.

As projects are hierarchical structures, access rights become inherited through subordinate projects so that, for example, the editor of a journal project also has editing rights to those journal articles which are subordinate projects in the journal. Such inheritance also extends to writing access and project maintenance which people in the subordinate project may not wish available. In such a case, the project can be simply 'de–coupled' from the parent project until it is ready for inclusion.

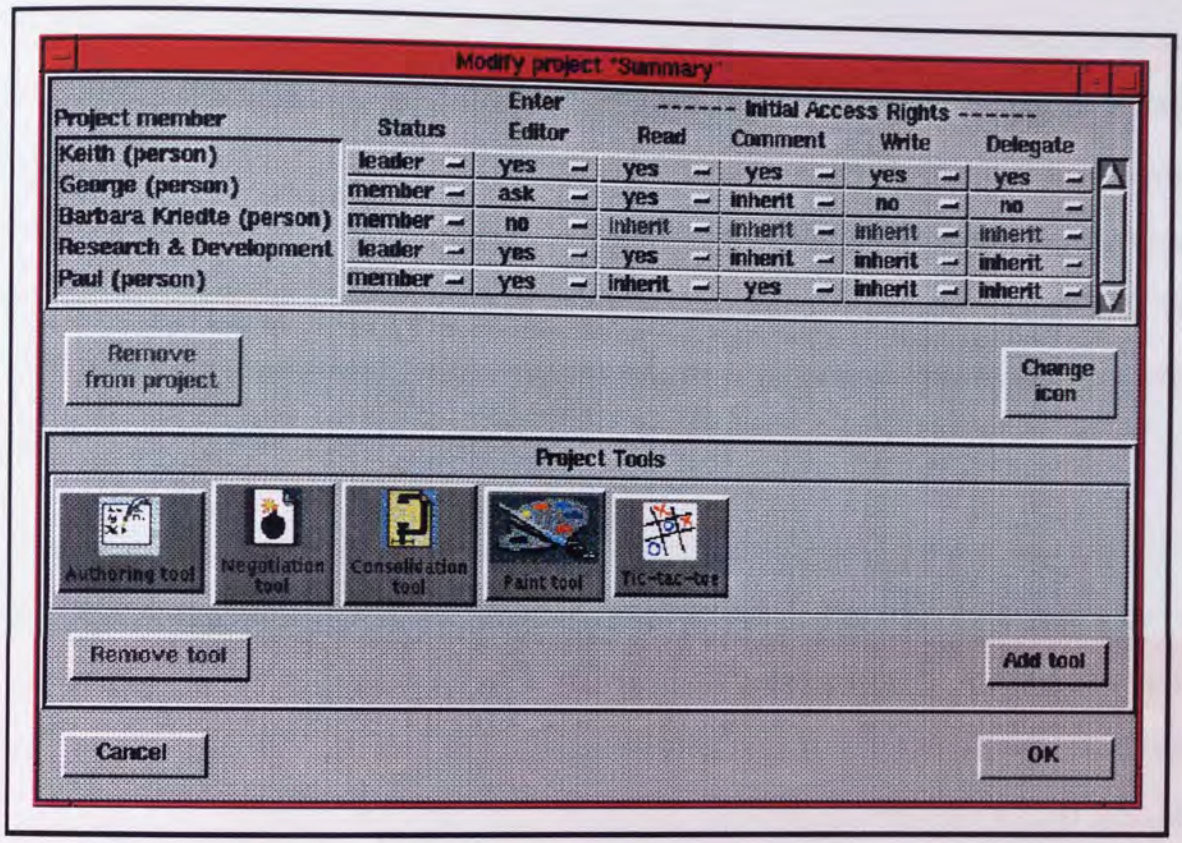

Figure 41: The Properties... dialog for a project

Figure 41 above shows the Properties... dialog for a project. This enables any of the project managers for the project to modify a number of attributes as they see fit. These modifications relate to those people who are able to participate in the project (the project members) and the tools that are available in the project.

6.4.1.3.a Project Member Attributes

Project members consist of people or teams. Each member can be either a member or leader, similar to teams, with the project leaders being responsible for the overall maintenance of the

project: explicitly adding or removing members, apportioning access rights to members and choosing those tools available in the project. If a team is made into a project leader, any team leaders in that team (or teams linked with it) are made into project leaders, with the team members being assigned as project members.

Project members are also each given a state as to whether they can join the authoring session. This state can be: YES (automatic access to the session), NO (no access to the session) or ASK (ask the project leaders whether access is allowed). If project members are allowed access to an authoring session, the rights they initially have in the session can be set for READ, COMMENT, WRITE and DELEGATE access. *Delegate access* enables a person to give access rights to another project member in the authoring environment, so that limited rights given initially may be changed locally as required. Each of these rights can be set as YES, NO or INHERIT. Rights inherited mean that the rights are determined elsewhere, and is a means of enabling specific positive or negative rights to be set in a project.

As an example, a team may be given only READ access to a project, yet it is wished that one of the team members be granted COMMENT access as well. In this case, the person's icon is dropped onto the project and his access rights set to: (`READ:INHERIT, COMMENT:YES, WRITE:INHERIT, DELEGATE:INHERIT`). If the team is later granted WRITE access, that person will also have WRITE access. To ensure a team member has only READ access to the authoring session, however, their individual rights could be set to: (`READ:YES, COMMENT:NO, WRITE:NO, DELEGATE:INHERIT`).

6.4.1.3.b Tool Attributes

Different tools may be added or removed by the project leaders for use in a project. By changing the tools, a project can be given different 'flavours' of use. For example, one project called "Diversions" could consist of groupware games, another called "Project Management" could consist of a groupware PERT chart and groupware conferencing tool, and another called "Chapter" could consist of the groupware authoring, negotiation and consolidation tools.

Figure 42 below depicts the dialog used for adding each tool, detailing the tool name, the command to run the tool and the icon to depict the tool.

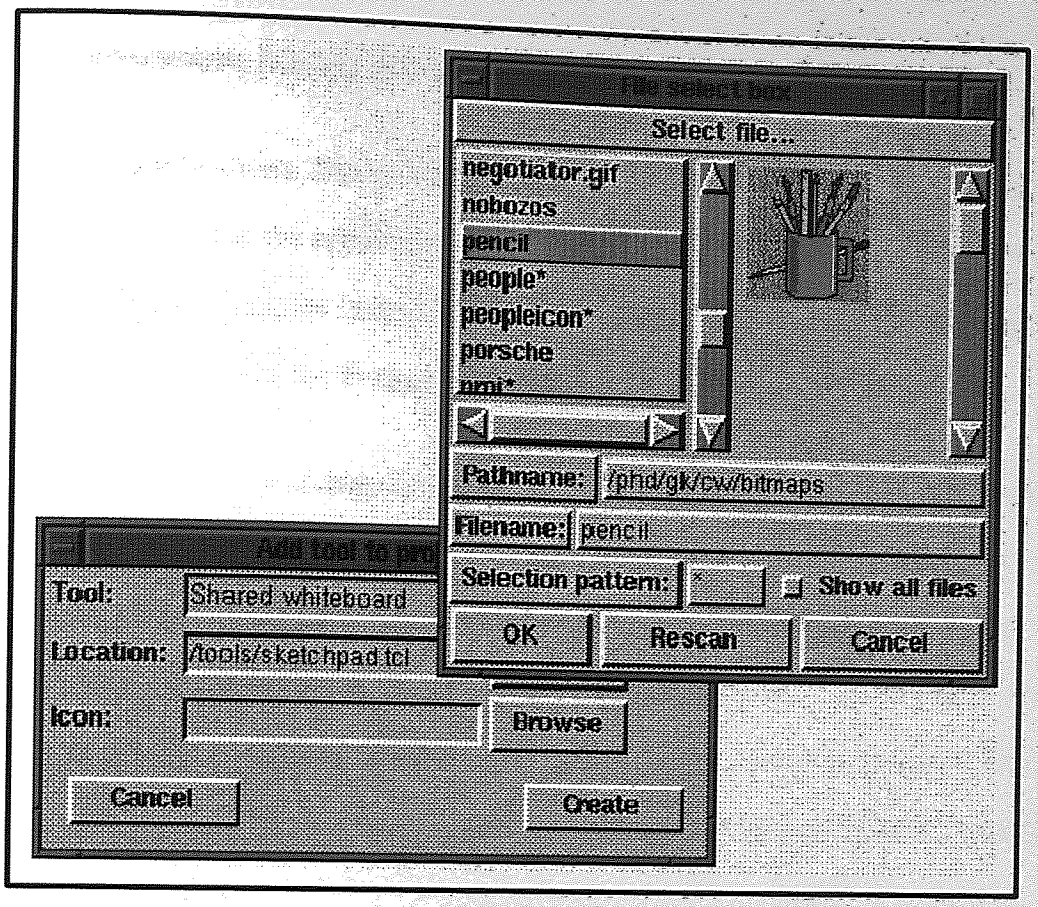

Figure 42: Add tool... dialog

6.4.2 Session Manager Technical Considerations

A number of technical considerations need to be addressed in the development of a groupware application. Many of these issues are addressed in the previous chapter, although there are a number of other issues dealt with in the Session Managers and which are also of relevance in the Collaborwriter conference processes.

6.4.2.1 Communications Method

As the session manager is not a conference process in a GroupKit sense but instead a method of *invoking* and *managing* conference processes, direct communications between session managers is not possible (as shown in figure 28 above). Instead, all communications are passed indirectly through the central registrar process using a client–server architecture for a shared environment variable (`collaborwriter`) with the session managers being clients

and the registrar being the server. If this environment is changed by one session manager all other session managers are notified of the change.

6.4.2.2 Groupware Dialog Boxes

In single–user applications, dialogs are often modal in nature as a simple way of ensuring that changes do not occur in the application which would make the information displayed in the dialog out–of–date. Thus, for example, Microsoft Word™ does not allow any changes to be made to a paragraph once the Format–>Paragraph dialog is displayed.

In groupware applications dialogs can be modal in the sense of modal to the local application but cannot be modal in a global sense. Thus, if one user opens a dialog displaying the list of people in a team that user might be temporarily restricted from dropping other people's icons onto that team. Opening this dialog should not, however, stop another user from dropping other people's icons onto the team. Such a dialog thus needs to be able to update itself if the environment it is portraying changes, resulting in dialogs not being modal in nature in both local and global cases.

In the session manager, such dynamic dialogs are controlled by using localised watches on the environment which are triggered whenever the environment changes. A typical code fragment for any such dialog is thus:

```
proc OpenThisDialog args {
   ...
   set dlogVars(bind1) [collaborwriter bind
      AddEnvInfo [list UpdateThisDialog vars %K]]
   set dlogVars(bind2) [collaborwriter bind
      ChangeEnvInfo [list UpdateThisDialog vars %K]]
   ...
}

proc CloseThisDialog args {
   ...
   collaborwriter delbind $dlogVars(bind1)
   collaborwriter delbind $dlogVars(bind2)
   ...
}
```

The update procedure `UpdateThisDialog` then needs to be written to use the current values found in the environment, and modify its current settings as appropriate.

6.4.2.3 Events for asynchronous use

A further difficulty with coding Collaborwriter compared with other GroupKit groupware tools concerns the potential asynchronous nature of a Collaborwriter session. Whilst current GroupKit conferencing tools use a model where 'latecomers' to a conference process are updated so that the data held is consistent with other processes, Collaborwriter often requires events to be generated between sessions which will lie dormant until the receiving user begins a session. For example, when a person asks to join a team this request is sent to all team leaders. If no team leaders are currently in the session, however, the request event remains active until the appropriate team leaders join the session.

Collaborwriter simulates such events using a portion of the collaborwriter environment listing outstanding events, the originator of the event, a list of receivers of the event and ancillary event information. When a user joins the Collaborwriter session and retrieves the collaborwriter environment information, the event data is checked to see if the user is in any of the recipient lists. If so, the event data is updated and acted upon. Thus both session data and event information can persist between different Collaborwriter sessions and different users.

6.4.3 Session Manager Voting Tool

Each session manager contains a simple voting tool for deciding issues between users. A typical issue is whether a team leader can be removed from a team, or whether a project leader can be removed from a project. In these cases each leader of the team or project concerned is asked to discuss and vote on the decision. Collaborwriter could also be easily extended so that other functions can be voted on, instead of having only arbitrary yes/no decisions by individuals (such as whether a person may join the team/project).

The Collaborwriter voting tool is a generic tool for determining the outcome of voting issues. It enables arguments to be either anonymous or public, voting by secret or public ballot and

decisions reached on an individual, majority or unanimous basis. A voting issue is created using the `CreateVotingIssue` command, with the format:

```
CreateVotingIssue
        [list of voters with current decision]
        [issue to vote on]
        [comments made on the issue]
        [command to execute on issue resolution]
        [list of choices available - e.g. Accept Reject]
        [voting method - unanimous | majority | individual]
        [whether peoples' names appear or are anonymous]
        [how the ballot is performed - public | secret]
        [voting event - e.g. newIssue | changeDecision]
```

The voting tool itself appears as shown in figure 43 below, depicting a request to remove user 'Keith' from a team with four members: Paul, Keith, George and Barbara.

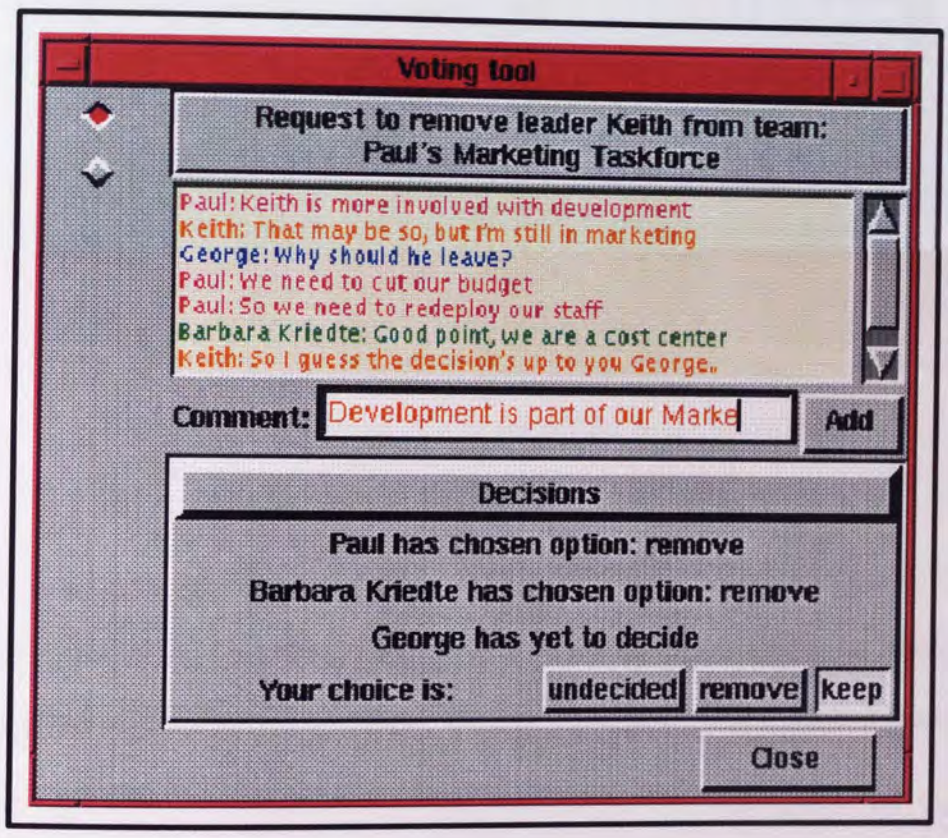

Figure 43: The voting tool as viewed by user 'Keith' for a public comment and public ballot issue.

The tool depicted above is displayed as viewed by Keith. For other users, the names and chosen option in the Decisions box would change as appropriate. Each decision–maker may comment on the issue, arguing a specific point, using their 'preferred user colour' and name

beside the comment. In an anonymous public ballot, the names and colours are removed from the Comments and Decision boxes, as shown in figure 44 below for the same issue, while in an anonymous secret ballot no reference is made to the way the vote is proceeding. Multiple issues may also require deciding on, and are independently selectable using the buttons along the left–hand–side of the dialog.

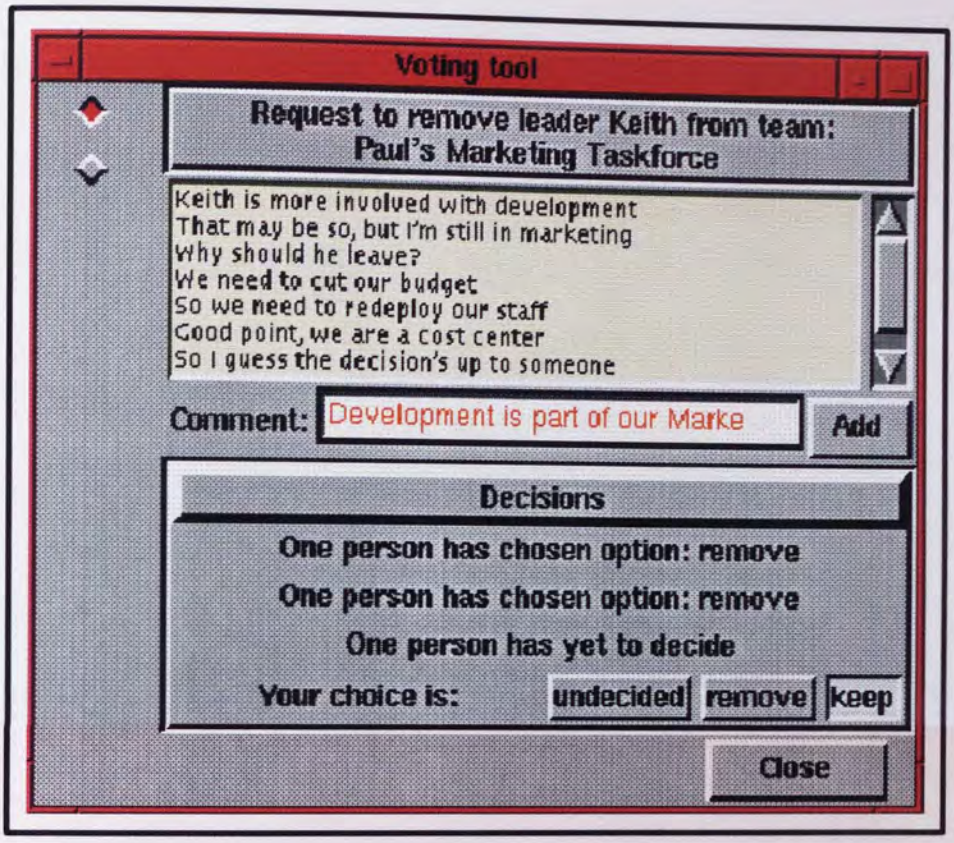

Figure 44: The voting tool as viewed by user 'Keith' for an anonymous comment and public ballot issue.

Once the decision resolution criterion has been met (with either an individual, majority or unanimous decision having been made) the voting tool closes the issue, displaying the decision taken and disabling any further comment/decision on the issue. Finally, the tool executes the command linked with the decision chosen.

Issues voted on are not removed once a decision is made, however, but instead are kept as an historical reference.

6.5 The Authoring Tool

The authoring tool is the main user application in Collaborwriter. Through it people can enter text, either synchronously or asynchronously, comment on others' work and link with the

negotiation and consolidation aspects of the work. The authoring tool is built on top of the policies designed in the Project Management session managers and thus adapts and changes focus as the session managers change.

The authoring tool can be regarded at four levels:

- the creation and management of document text nodes;
- data synchronisation;
- the creation of text inside a node; and
- comments and annotations on current text.

6.5.1 Text Node Creation and Management

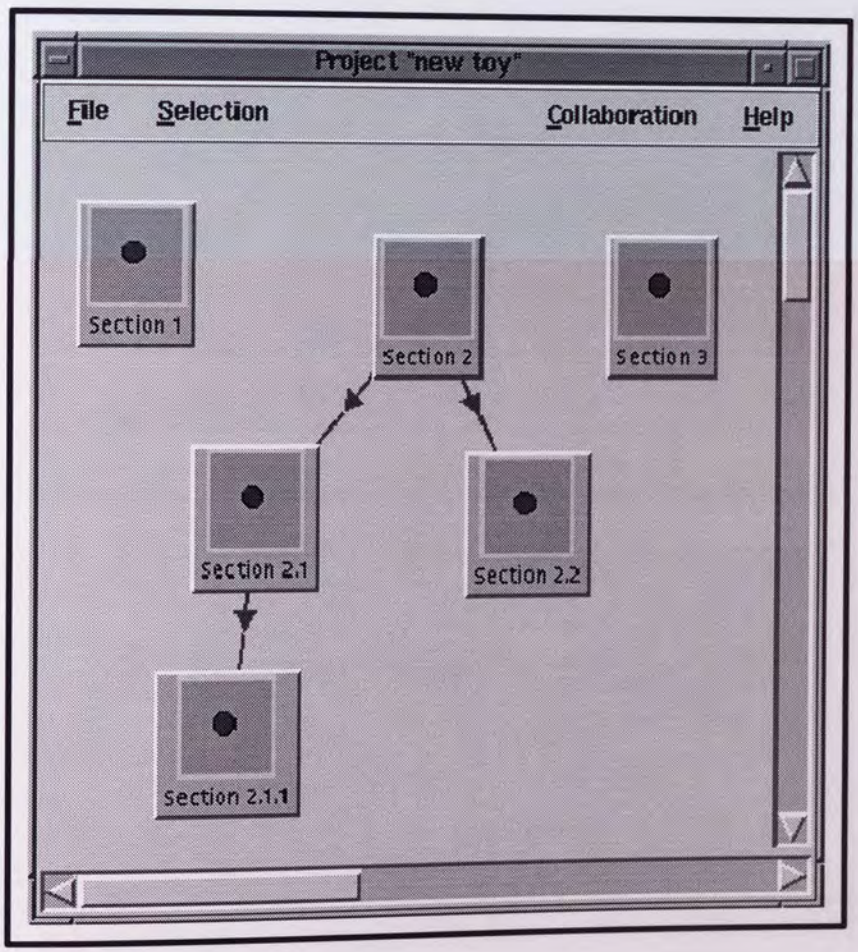

Figure 45: A typical document node hierarchy.

Collaborwriter uses a hierarchical document model for text creation, similar to the document model used in SGML (Goldfarb, 1990). Figure 45 above shows a typical document node

hierarchy, with the document created consisting of three top–level nodes (Sections 1 to 3) and sub–sections and sub–sub–sections for section 2. Such a hierarchy is equivalent to the Front Matter, Main Body and Back Matter in a published book.

New nodes are created by simply double–clicking on the workspace. The new node may subsequently be linked to a parent node by dragging the new node over the parent icon. The names used for the current nodes are also automatically recalculated when new nodes are created and linked with, for example, Section 1 becoming Section 2.1 and Section 2 (and its sub–sections) becoming Section 1 (and sub–section 1.1 etc.) when Section 1 is made a child of Section 3 (see figure 46).

Nodes may be moved, tidied and have their links broken in an identical manner to that used by the session manager.

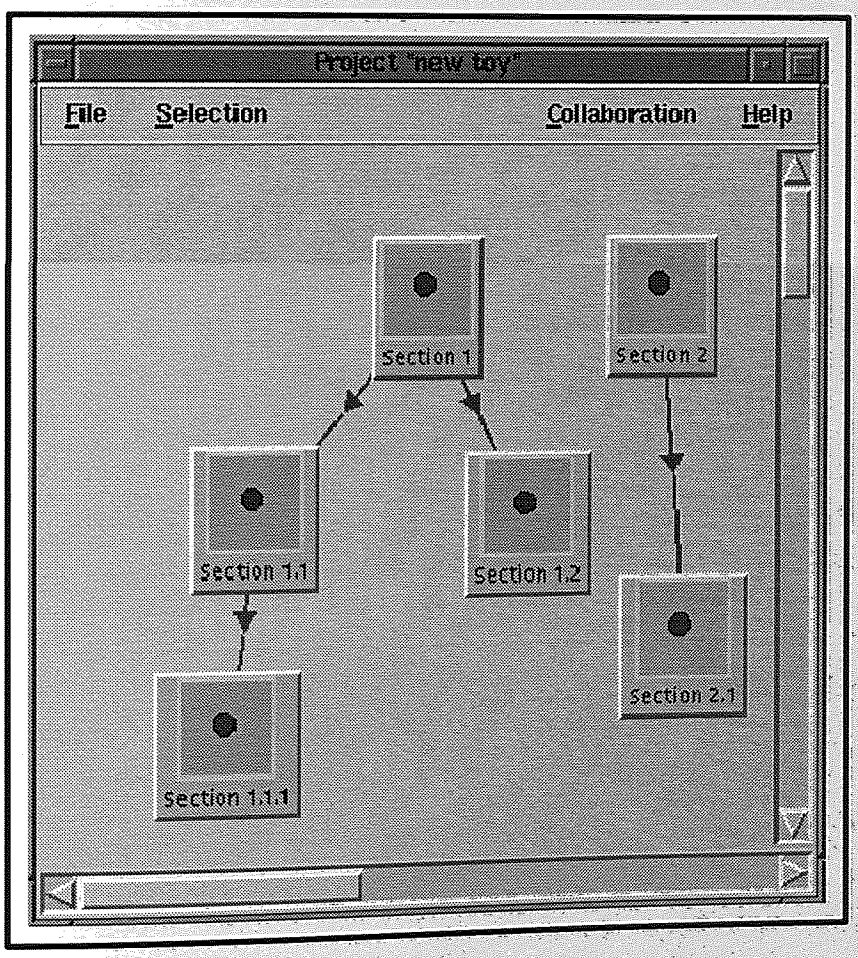

Figure 46: Automatic section naming in the document node hierarchy.

To aid author awareness, each node contains a GroupTable widget, showing quickly and easily who is currently accessing a particular node or group of nodes. Access to particular

nodes is managed as an extension to the session manager access protocol, with authors only able to give access to those users present in the project 'world' as defined in the session manager. These access rights are inherited by any child nodes with more specific privileges able to be assigned at the local node level. Figure 47 below shows the Properties... dialog for an authoring node.

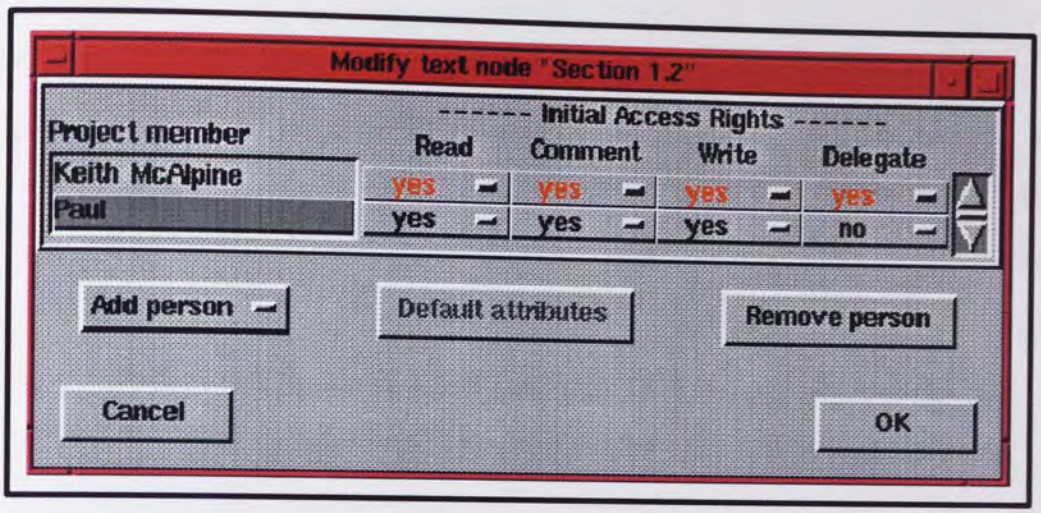

Figure 47: Properties dialog for an authoring node.

This dialog uses black text to depict access rights which have been inherited from a parent node and red text to depict access rights which have been added or changed locally to that specific node. By choosing the "Default attributes" button changed attributes may be reverted to those values defined in a parent node. All changes made in this node also get passed recursively to any child nodes.

Thus the access rights to the authoring tool are initially managed through the session manager, which determines the set of people visible to the authoring tool, along with their initial access rights in the tool. Authors can then determine further tailoring of access rights within the tool as deemed necessary.

As an example, a project node could be created with two users given full delegation and authoring access in the tool and a further ten users given read access in the tool. Internally chapters or sections may be created by the two originators and the other ten users then assigned various roles and responsibilities over these nodes (authoring, commenting and

delegating). Particular users may also be excluded from reading a specific section (and its sibling nodes) by applying a negative access right to those users.

6.5.2 Data Synchronisation

Text created in a node is not kept 'up–to–date' between all users at all times, but is instead replicated on a 'need–to–know' basis. Thus a form of *delayed synchronisation* of data is used.

When a user enters a node they ask for the latest version of the text in the node. They may then make any changes as appropriate to the text in that node. When they leave the node any changes made are stored in the document node database. Where multiple users are present in a node, however, tight collaboration is automatically invoked with any changes made being sent automatically to those people currently viewing the node contents. This way, people interested in the node have the most up–to–date information available inside the node, whilst others will not be informed of any changes being made.

When a node is created or a node link changed, however, all users in the current session are automatically informed and their displays updated. Users may also request specific nodes to have tight collaboration, where any dragging or changes to the node itself will be replicated on any interested parties' displays.

6.5.3 Text Entry

The text entry portion of the authoring tool is where the text for each individual node gets entered. It is built from a number of widgets (multitext, attrpicturebar and group scrollbar) described earlier and is depicted in figure 48 below.

For localised author awareness there is a modified version of the PictureBar widget, called an "AttrPictureBar" which, as well as depicting each of the authors graphically also contains radio buttons underneath each author graphic which details that person's access rights in the node (whether read, comment, write and/or delegate). On the side of the display there is a GroupKit group scrollbar showing the relative positions of any user in the node, along with the ability to shadow a particular user as they move around the document node.

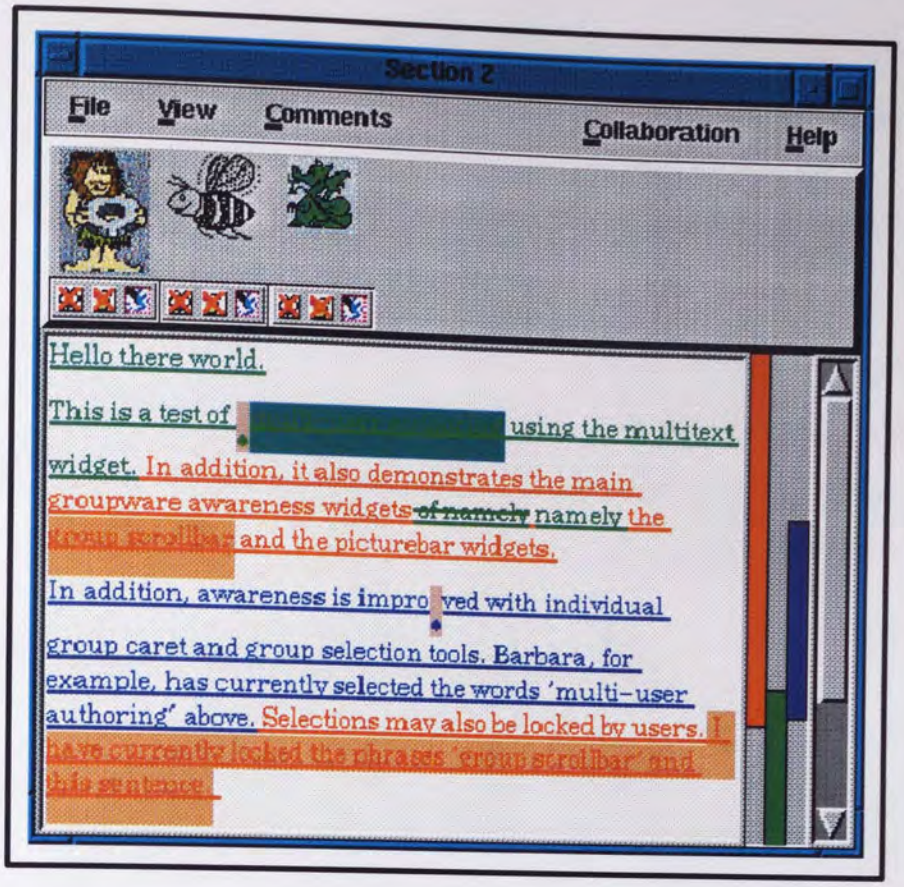

Figure 48: Text entry in the authoring tool with three users present.

Awareness is further reinforced in the multitext widget, where text inserted is written using the author's colour, deleted text appears in a darker shade of colour, and locked regions are shaded in a lighter variant of the locking author's colour. A marker is also shown depicting which authors are typing on a specific cursor position in the document. This use of colour may lead to reading difficulties, however, especially when changes have been made by a number of authors, so there is an option to locally remove the colour from the text entries, using a shade of grey for deleted text and black for inserted text. Colour remains for the user markers and locked regions feedback.

Direct communication between authors is achieved by clicking on the appropriate author bitmap, causing a directed message (e-mail) to be sent to that specific author. Broadcast messages are sent by extending the selection and clicking on other bitmaps. In both cases, the system maintains a record of such messages in order to form a history should conflicts arise or negotiations become necessary.

If an author has delegate capabilities, they can change the access rights of an individual author by toggling the appropriate access right radio button below the person's bitmap image.

Finally, to aid authors in structuring their text there is an option to use the negotiation tool as a *pattern note creation system*, either privately or as a group memory. In this mode, the negotiation tool allows authors to create new link types in addition to the standard types described later. It does not enforce any rules on these types, however and, by using a shared whiteboard, the negotiation tool allows the user to make any type of pattern note web.

6.5.4 Comments and Annotations on the Text

The authoring tool also enables authors to add comments to previously written text. A comment may be applied to a specific word, paragraph or section, or to the whole text node.

A comment is simply created by making a selection on the text and choosing "Add comment..." from the text editor menus. If no selection is made the comment applies to the whole node. To apply a comment to a range of nodes (section 1.1 and below, for example) the comment should be created on the text node view and called using "Add comment..." from the node menu.

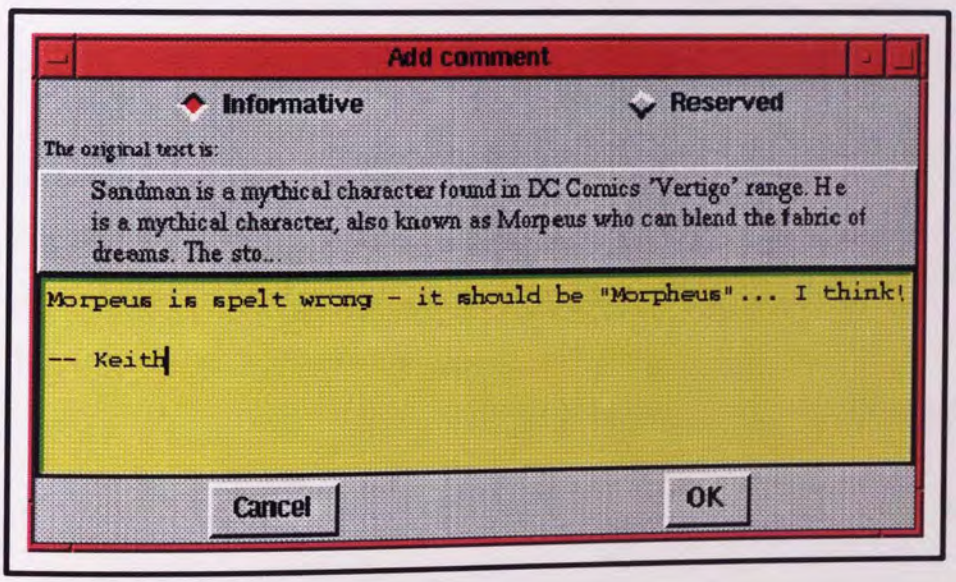

Figure 49: The Comment dialog from the Authoring tool.

Figure 49 above shows a typical comment dialog in the authoring tool. An author may choose to make either an informative or a reserved comment on the text. In the case of an informative comment there is no potential for conflict, the comment being simply a suggestion or advice by the annotator. These comments are implemented as PostIt ™ notes and contain the original text being commented on along with any messages. This comment is positioned at the start of the text and may be deleted by the originating annotator or by any valid author for that node.

When a reviewer makes a reserved comment there is a potential for conflict. This then links with the negotiation tool which is based on a free–form issue–based system (Rittel and Kunz, 1970). Where no reserved comment has been previously made for the text, an "issue" is automatically created. Otherwise the reviewer (or an originating author) may choose to add or include an additional position, argument or issue note based on the original reserved comment issue. The resolution of such comments are detailed in the negotiation tool section.

Finally, authors may wish to use an unstructured shared whiteboard as a form of scratchpad for eliciting ideas and suggestions. This feature is also incorporated in the negotiation tool. Subsequently the persistence of such unstructured information will depend on whether it has been created as the result of an informative or reserved comment.

6.6 The Negotiation Tool

Along with the authoring tool, the negotiation tool is the other central application in the Collaborwriter environment. It is based heavily on the IBIS system (Rittel and Kunz, 1970; Conklin and Begeman, 1988) of structuring information for discussion and debate, enabling authors to see what conflicts are open and other people's current position on those conflicts at any time. In addition, unstructured information may be added through the incorporation of a shared whiteboard (Stefik et al., 1987) within the tool. The negotiation tool therefore enables authors to work through structured arguments, to be used as a scratchpad for informal or unstructured notes, or be used with a mix of both methods.

The negotiation tool is opened either directly from the session manager or indirectly from the authoring tool through the creation or review of either a reserved comment or a pattern note authoring aid.

When examining the negotiation tool, it can be divided into four main areas:

- the IBIS structure;
- the shared whiteboard;
- hierarchical issues; and
- pattern notes.

Each of these areas are described below.

6.6.1 The IBIS Structure

The IBIS structure is based on a set of interconnecting nodes that form an argumentation system. The nodes can represent either issues, positions or arguments, and may be joined through a defined set of typed links. Subsequently a conflict becomes structured to enable one to see all the issues relating to the conflict, what other people's positions are in relation to the issues, and the arguments for and against those positions.

To represent this structure, the negotiation tool has at its core the IbisIcon widget. Tables 8, 9 and 10 below summarise the possible links when linking to an issue, position or argument node respectively along with the textual and iconic representations of the links as used by the IbisIcon widget.

If linking to an ISSUE...		
New issue	generalises	current issue.
	specialises	
	replaces	
	questions	
	is–suggested–by	
New position	responds–to	current issue.
New argument	*not possible*	

Table 8: Possible links when joining to an issue.

If linking to a POSITION...		
New issue	questions	current position.
	is–suggested–by	
New position	*not possible*	
New argument	supports	current position.
	objects–to	

Table 9: Possible links when joining to a position.

If linking to an ARGUMENT...		
New issue	questions	current position.
	is–suggested–by	
New position	*not possible*	
New argument	*not possible*	

Table 10: Possible links when joining to an argument.

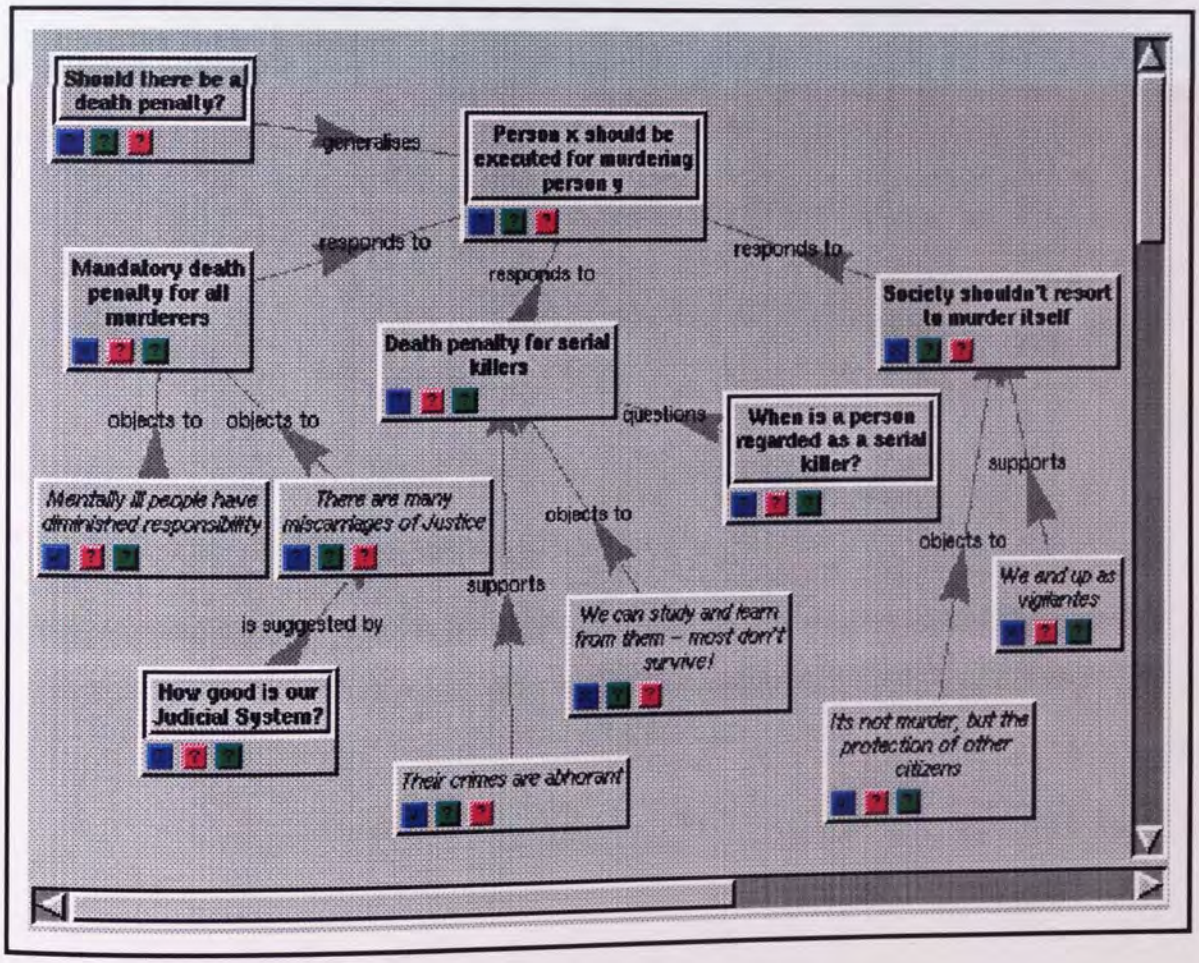

Figure 50: An example of an IBIS argumentation network.

Through the enforcement of these typed rules by the system, participants become required to structure their arguments which results in a clear argumentation pattern being formed. In this manner, people may discover that the issue related to a specific conflict is not, in fact, the real problem but that another related issue is at the core of the conflict.

As an example of a conflict, figure 50 above depicts an IBIS node network created by three people where the initial issue was "Should person x be executed for killing person y?" From this figure it can be seen that the issue may be generalised ("Should there be a death penalty?") or, as a result of the participants' positions and arguments, new issues raised ("The Justice System", "What represents a serial killer?", etc.)

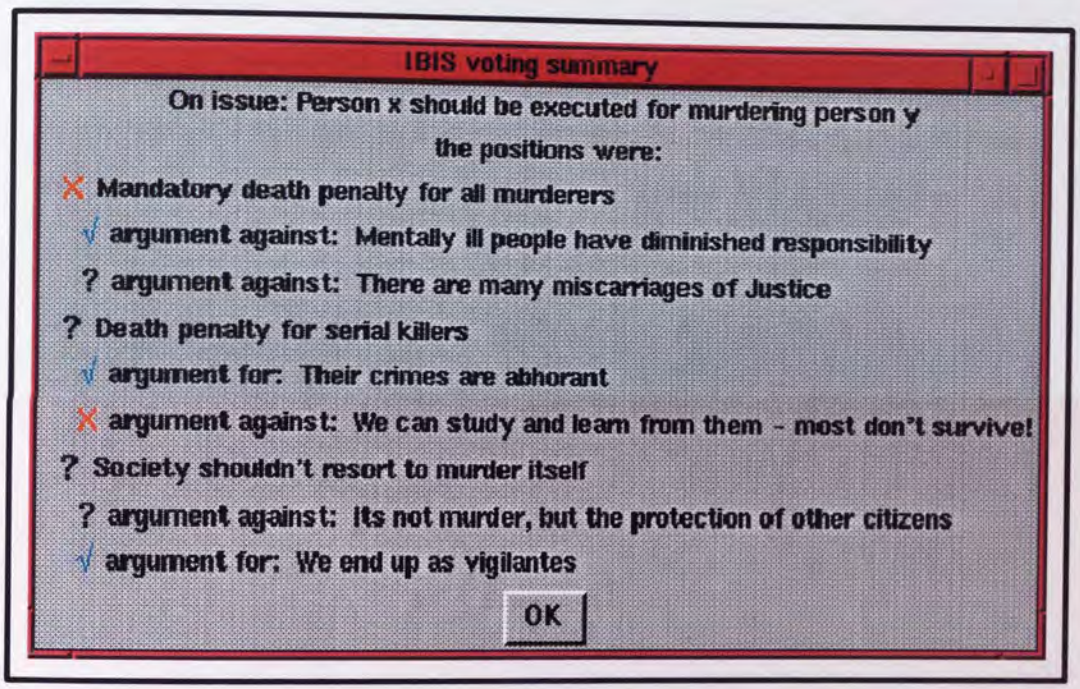

Figure 51: Voting summary on an IBIS issue.

In addition to helping structure one's arguments, the negotiation tool also acts as a voting tool. Each participant has a coloured button underneath each IbisIcon widget which may be changed to a ✓ for agreement, **X** for disagreement, or **?** for undecided on an issue. At any time a person may request what the current consensus is regarding an issue, position or argument. Collaborwriter then tallies the votes cast and presents the preliminary results in a dialog. When a vote is called, a message is sent to each participant asking whether they are satisfied with their choices and what is their preferred voting method (majority or unanimous). This

request is persistent over sessions so the vote may be cast asynchronously in a similar manner to that used by the session manager voting tool. Once all votes have been cast a summary is given of the preferred position to a particular issue, and what arguments were agreed or disagreed with. Figure 51 above shows an example summary based on the death penalty issue presented above.

6.6.2 The Shared Whiteboard

In addition to managing structured arguments, the negotiation tool may also be used as a freeform shared workspace in the manner of a shared whiteboard (Stefik et al., 1987). This workspace is the same as used for the IBIS nodes with the added functionality of a group of drawing icons represented on a palette along the bottom of the screen. Figure 52 shows the drawing tools available, which consist of simple graphics primitives (paintbrush, line, box, circle, text) and a special option to insert the text from the authoring tool which is under comment.

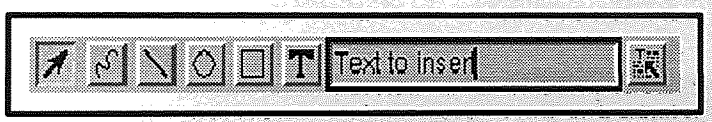

Figure 52: The drawing tools available from the shared whiteboard.

The colour used for drawing is each author's chosen colour, consistent throughout the whole of the Collaborwriter environment, except for where the original text is inserted which is instead drawn in black using a large bold font. This method simulates having a transparent window over the text for annotations, including an option to update the underlying text with the current contents of the authoring tool. Marks made pointing to specific words may become invalid, however, depending on what form the updated text takes.

As the shared whiteboard uses the same display as the IBIS creation, both may be mixed so that informal notes may be written on the background of the canvas used to display the argumentation networks. An example of merging both systems is shown in figure 53 below. Both elements may not be linked, however, so moving an IbisNode widget icon will not move any text written on the canvas surrounding it.

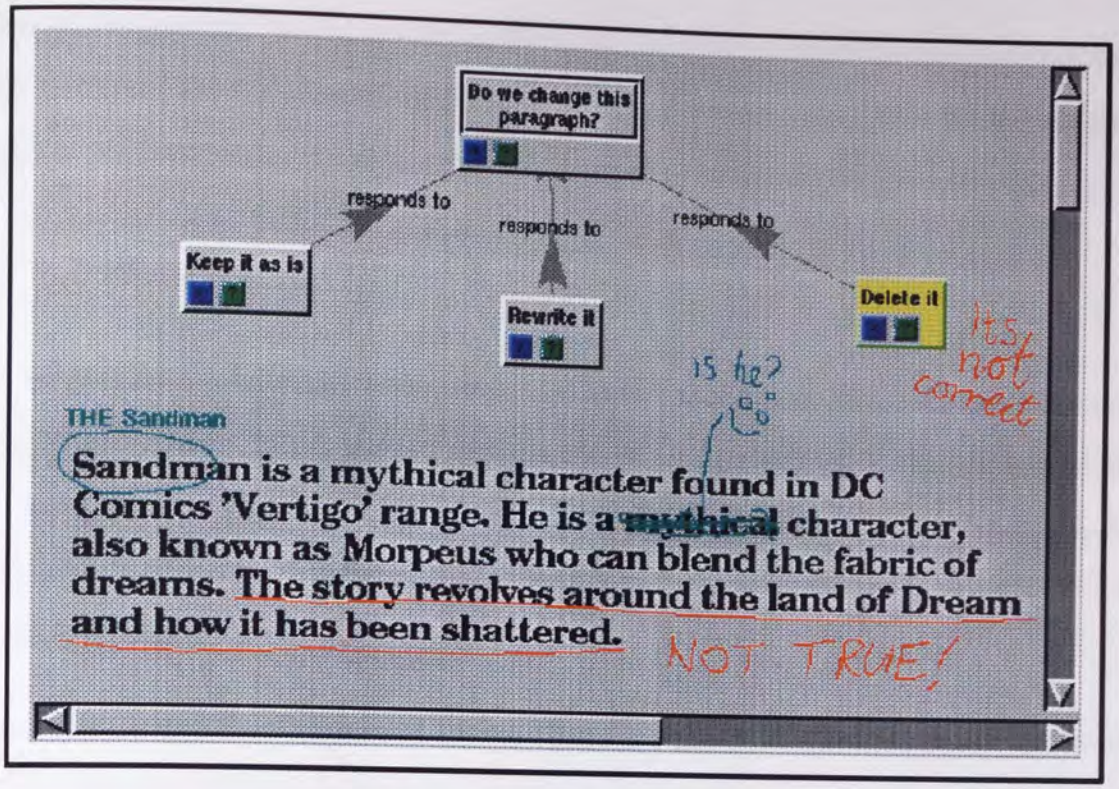

Figure 53: Screen showing a mix of structured and unstructured argumentation.

6.6.3 Hierarchical Issues

The negotiation tool makes use of the hierarchical nature of the document for structuring and ordering the display of issues. Where an issue affects a complete node or range of nodes, that issue is portrayed as a bar to the left of the node name(s). Where the scope of another issue is smaller, it appears to the right of the original bar, closer to the node name. Issues internal to a node (based on a paragraph, for example) appear to the right of the node name, with issues again smaller in scope represented to the right of the original. Figure 54 below shows an example of this hierarchy.

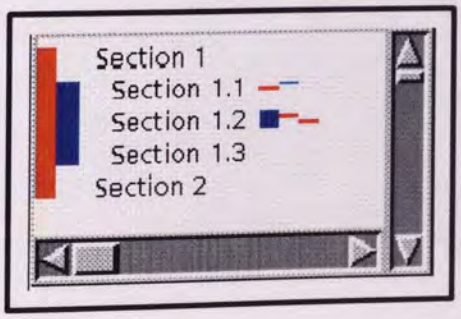

Figure 54: The hierarchical nature of issues.

Using this representation authors can see the scope of any of the issues needing to be resolved, with those with the widest scope (and potentially biggest impact) further to the left and those with the smallest scope (possibly only a single word or sentence) further to the right.

6.6.4 Pattern Notes

The final element of the negotiation tool is the ability to create pattern notes (also known as *mind maps* – Buzan, 1983). Pattern notes consist of pictorial representations of information linked by association. For this, the author may choose to use either the whiteboard drawing elements for creating the pattern notes or the IbisNode icon widgets with a custom link for formulating ideas.

The pattern notes implementation differs from the argumentation implementation in that authors are not restricted in their use of colour for drawing, nor are they restricted to the pre–defined set of IBIS link types. The three entity types (issue, position, argument) remain, with it being left to the author to decide how to interpret them, although the voting element of the widget is removed. Because of these differences, the user must either switch to "Pattern Notes" view using the negotiation tool view menu or explicitly call the negotiation tool by requesting to create a pattern note from within the authoring tool.

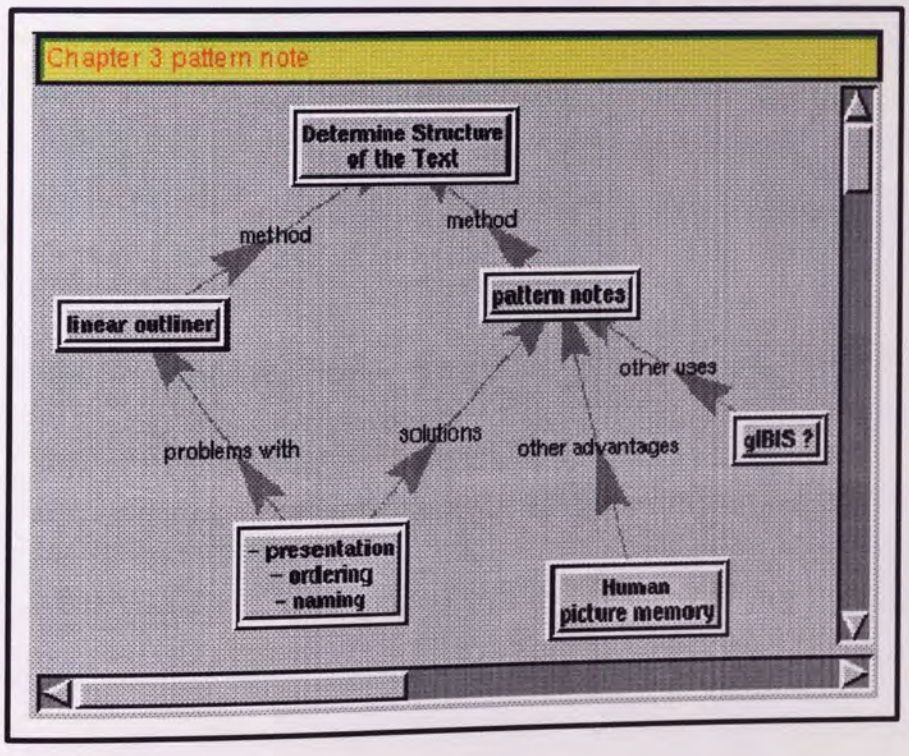

Figure 55: Pattern note created with structured methods viewed via the negotiation tool.

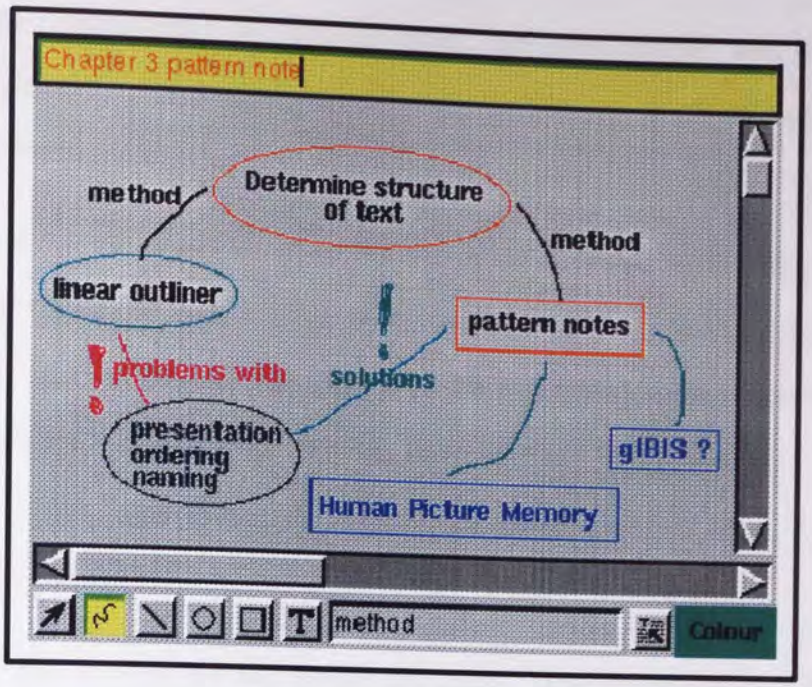

Figure 56: Pattern note created with unstructured methods viewed via the negotiation tool.

Figures 55 and 56 above shows the pattern note representation on the system for the same pattern note created in figure 7 of section 3.2.1. This has been created using both structured and unstructured methods in the negotiation tool.

6.7 The Consolidation Tool

The final element of the Collaborwriter environment is the consolidation tool. This is actually implemented inside both the authoring and negotiation tools. In the authoring tool the multitext widget is used to control the version of the document while the negotiation tool makes use of the hierarchical issues portion of its system.

Starting within the negotiation tool, and beginning with the right–most issues, the system will check if issues created have been resolved by a vote. If not it will request a vote to determine whether a decision can be made. If the issue is resolved it is logged and removed from the issue hierarchy. In addition the issue placeholder is removed from the node or textual element in the authoring tool from where it was linked. If the issue is not resolved it is kept in the system to be resolved at a future date. After each consolidation pass, however, the issue bar changes colour to denote that it is an outstanding conflict, changing from black to dark red to bright red. This enables both current and new authors to see any particular 'sticking points' which still need to be resolved, along with the scope of the document to which these points refer.

In addition to consolidating outstanding issues, the consolidation tool merges levels of created text (the editing sessions of the multitext widget) into a single session, thus creating a new baseline for the text.

After consolidation, authors may choose to negotiate their new roles using the negotiation tool linked to the appropriate section nodes. Access rights to the various nodes may be modified. The new author roles will then be visible as a memory cue to those working in the nodes and represented as an issue spanning possibly all of the document.

6.8 Conclusions

This chapter has focused on the development and implementation of Collaborwriter, a groupware tool to aid collaborative authoring. It has examined and described the main components of the system: the session manager, the groupware widgets and the authoring, negotiation and consolidation tools. These tools have been designed with the writing model formulated in chapter 4 in mind, and using many of the techniques outlined in chapter 5. In addition it has described some of the technical difficulties in developing such a system which are unique from the general difficulties inherent in developing a substantial software system.

The following chapter will conclude this thesis with a description of user feedback based from some preliminary trials of the system, detailing some areas where the system could be improved, and giving pointers to future research in the area of collaborative writing.

Chapter 7

User Trials, Conclusions and Future Work

This chapter is split into three main sections which bring this thesis to a close. The chapter first reports on some preliminary user trials with Collaborwriter, a multi–user authoring environment described in the previous chapter. It then draws some conclusions which can be made from the development process of the system and with regard to collaborative applications development in general and collaborative writing in particular. Finally, the chapter concludes identifying areas which need to be addressed for further research.

7.1 User Trials

In order to assess the usability and acceptability of Collaborwriter preliminary user trials have been carried out. These trials were of an informal nature, running over short periods of time and with between two and five users.

Three types of trial have been carried out:

1. trials using only the authoring portion of the tool;

2. trials using only the negotiation portion of the tool; and

3. trials using session management, negotiation and authoring aspects of the tool.

All the users in the trials had a fairly comprehensive knowledge and experience of using computers and single–user word–processing systems.

The trials took place in a single room which consisted of three computer systems. Partitions prevented participants from physically viewing one another's terminals although verbal communications were permitted being analogous with a telephone conference–call.

7.1.1 The Authoring Tool

Two trials took place with the authoring tool. These trials were all aimed at examining the node creation in the tool, awareness aspects of the tool, the locking mechanism in the editing part and the users' perceptions of the tool's general value.

7.1.1.1 Two users working synchronously in a single text node

The first trial consisted of two users working together in a single node. They were asked to produce a simple one page description on some of the satellites developed at the European Space Agency.

Both participants were situated in the same room using separate computers. Verbally they discussed what satellites they would include, writing their names on separate paragraphs. Each wrote about a separate satellite, although both would stop at intervals to read what the other had done and make comments on it with gesturing in the text performed by using the highlighting mechanism. Neither participant wrote in the others' text, however, simply suggesting what the other should write or change.

After the trial both participants agreed that the awareness aspects of the tool were good with respect to the scrollbar, text highlighting, remote carets and telepointers. They did not see the value of the picturebar, however, and would have liked an option to hide it in favour of more writing screen area.

7.1.1.2 Three users working synchronously and asynchronously in multiple text nodes

In the second trial three people were involved in writing a more involved document. In this case the group could choose what to write about, with the document required to be finished by the end of the day. The people could choose to work synchronously or asynchronously on the document, being able to enter and leave the laboratory containing the computers running the program at any time through the day.

Initially the group met to decide what to write about and assigned some division of work to each member. This involved creating the basic parent–child nodes and links for each division

– a process performed by the group with one user working on a single machine and the others suggesting ideas. At this stage the group wanted to be able to change the node names from the automated "Section 1", "Section 1.1", "Section 1.2" and "Section 1.3" to the titles "Computer Games", "Arcade", "Adventure" and "Strategy" respectively. On discovering this was not possible the members positioned the nodes spatially on the screen and agreed to use alphabetical order for determining each node's content.

The group then dispersed, being left to choose individually when and how often they would return to work on the system. They agreed, however, to all meet again an hour before the end of the day to check and consolidate the completed work. Table 11 below shows the frequency and length of time each member returned to use the system.

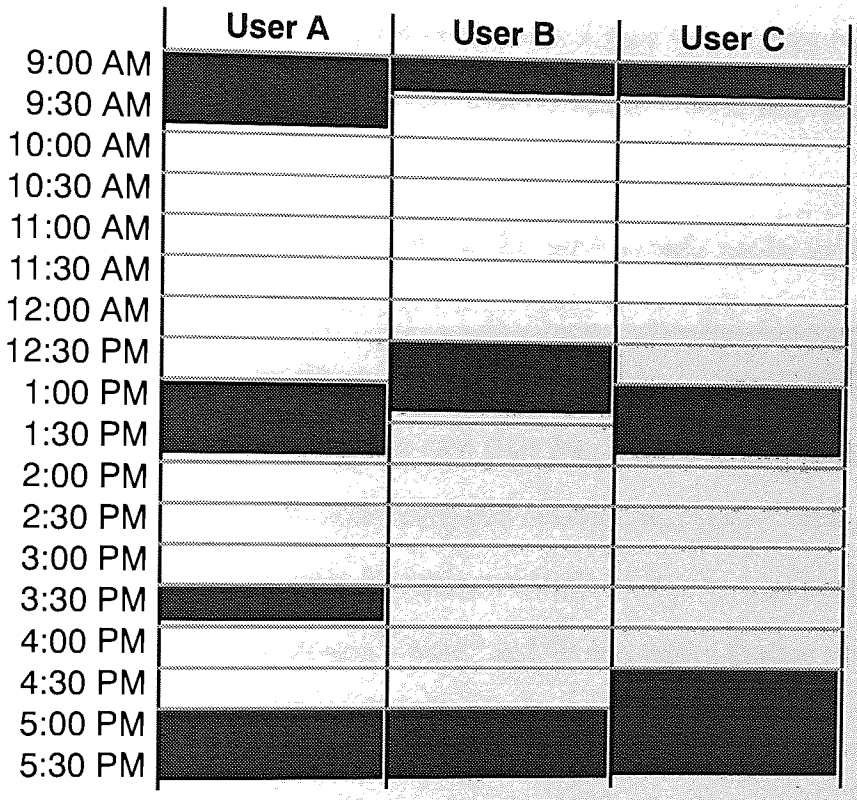

Table 11: Pictorial representation of the time and frequency for three people writing throughout one day.

As the experiment took place in a working environment people chose to work at similar times, using the system mainly over their lunch period and at the end of the day. This resulted in all the users working on the system at 1:00 PM for approximately half an hour and between 5:00 PM and 6:00 PM for an hour.

It was observed that all three users would initially open all the text nodes when starting a session in order to see if any changes had been made in the nodes. If a node did not concern the user or no other participant was working in it then the node was either closed or minimised to reduce screen overhead (the system was running using two 14" and one 15" monitor). One person remarked that it would be useful to have some mechanism for seeing where or what changes had been made prior to starting the session, either through a history log or some visual feedback.

During the test it was also observed that when multiple users were present each person had a tendency to simply observe another user working. Thus a person would first read what others had done and then iteratively write some text followed by watching the others again. Indeed one person had the "follow this user" option enabled for different users in the two other nodes. He could thus simultaneously watch the other users working in their individual nodes.

When wishing to comment on another's work, the users simply spoke to the appropriate person, using the highlighting mechanism for gesturing to the text concerned. When the person was not present, however, the users made comments by creating a new paragraph and entering the text. Talking afterwards, they said they found the comments more noticeable when entering the text direct in the node compared to using the PostIt™ style of unreserved comment made available in the system. The users were reluctant to manipulate the other's text directly, saying "It was the other person's work" and "It was easier to remove the comment paragraph totally than multiple changes in a paragraph".

At the end of the day the three people met for an hour to finish and consolidate the work they had produced. For this process the group read everything which had been written previously, commenting verbally aloud on the text and using either the telepointer or remote highlighting to gesture towards the item of text. At this point only one user made changes to all the text nodes. It appears that people were reluctant to change each others' work while they were not present, regarding it as the individual's work while, when working synchronously, the text was regarded as the group's work and open to changes by any of the members.

None of the participants chose to use the locking feature present in the tool, each feeling there was no need as each member informally 'owned' the node they were working on, even though all members had equal access rights to all the nodes. The group felt that a larger, more formal group, working over a longer period of time, would need to use both the comment and locking features of the tool.

One member said that under different circumstances he would actually use both locking and comment features together, locking a portion of text and then creating a comment visible to all explaining what the text represented or meant. He felt that it was important for an author to be able to communicate with others directly in the text and that the mechanism would be used not to lock the text but to bring it to the attention of others entering the node at a later stage. As colour was used to identify users and fonts had no meaning in the tool (each user could set their own preference for how to display the text) he felt this was the only mechanism available for highlighting text.

Discussing the tool with the group after the trial, all the members found the process to be "enjoyable" when other members were present and "satisfactory" when working in isolation. They said that as the tool had no layout and formatting tools they were unable to do anything other than write when they were alone. In the group setting, however, the members were able to follow another's text and comment on it as it was being written.

When asked if they felt the tool would be useful in a larger scenario all felt that both import and export functions to other word processors would first need to be available so that one could work "off–line" on another system and subsequently paste the text into the tool. An example was the ability to import and export to a single Microsoft Word™ document with the node levels in the authoring tool converted into Heading level styles in Word and vice–versa.

With regard to the groupware widgets the group said that they found the grouptable widget useful for identifying where people were working. In their case it was not very relevant as all the members opened all the nodes when starting a session, even if they were only working in one. When scaling up with more users and many more nodes they felt this would be a useful element of the system. The group would have liked some text formatting controls available

within the multitext widget but accepted that this was a trade–off against the awareness aspect of the widget. In that respect the members felt that the split between the node view and the text views for user awareness worked well, but they would have liked to be able to change the node titles.

Overall the people participating in the trial thought that the synchronous capabilities of the authoring tool were impressive and wished the elements in it were available in more mainstream word processing systems.

7.1.2 The Negotiation Tool

Two trials took place to assess the negotiation tool, and were aimed at examining the IBIS argumentation model for structured argumentation and the shared whiteboard model for unstructured argumentation. Three people took part in the trial and both trials took place synchronously. Again, the users were in the same laboratory so they could talk to one another although each worked on a separate computer.

7.1.2.1 Three users working synchronously with structured argumentation

In the first trial three users worked together using the IBIS argumentation methodology for representing their position and arguments for and against that position. The group was asked to discuss and argue over an emotive issue – "Should person x be executed for murdering person y?" Although the group size was small, the members had strong conflicting views on the issue and had to argue their positions either for or against it.

In use, the participants initially had a difficult time with the terminology in the IBIS method. The group first formed two positions, "yes" and "no" and put arguments for or against each position. They then decided this was the wrong way of doing it, as each argument which "supports" one position would "object–to" the other position.

After some discussion, the group decided to generalise the issue to one of whether there should be a death penalty and chose to use *statements of position* which would determine the members' points of view. This consisted originally of two opposing positions: "Mandatory death penalty for all murderers" and "Society shouldn't resort to murder itself".

During the discussion, the members used the telepointer facility as a means of gesturing to a specific node. Before creating a node, the members tended to discuss first what it would contain. For example, one member verbally stated that people would end up as vigilantes if they resorted to executions. While this statement was not agreed with by the other members the originator would still include it in the argumentation net with his vote on it.

As the discussion progressed another position became clear and other issues were raised. At its conclusion, while none of the group members had changed their own views or positions, they appreciated that other positions and issues were involved. The members found this visualisation of the argumentation helpful and thought it brought out certain issues which they would not have originally considered. They found the choice buttons useful not for voting on an issue but for easily identifying what each member thought about a particular issue.

On a final note, however, the members felt that the wording in a node was of key importance. One member wished to object to the argument "We end up as vigilantes" with the counter–argument that controls were in place to prevent such actions. The IBIS scheme does not have an option for formulating arguments for or against other arguments. Instead, an issue would need to be created that questions the argument and further positions and arguments formulated around that issue. The participant simply chose to vote against the argument, not including *why* he was voting against it.

7.1.2.2 Three users working synchronously in unstructured argumentation

The same group was then invited to discuss an issue on whether violence on television leads to increased violence in society or whether it simply reflects that society. In this case the members were able to use both the whiteboard and the relaxed IBIS tool, with the ability to create new links at will between nodes.

Similar to the first case, the group had difficulty finding a starting point for the negotiation, first choosing to create an issue "TV and Violence" and wrote "YES" and "NO" directly onto opposite sides of the screen as list headings. They then changed this by creating two positions – "Causes" and "Reflects" – linked to the original issue. The members reverted to the drawing tool, choosing to list reasons for each position underneath the respective position boxes.

From this trial it was discovered that the members all had similar views on the issue and that, instead of being an argumentation, the session became a brainstorming exercise in determining possible reasons for and against the central issue. Discussions on the listed items took place verbally with the freehand tool used to strike–out or emphasise specific items. Figure 57 below depicts a screen shot on the results of the exercise.

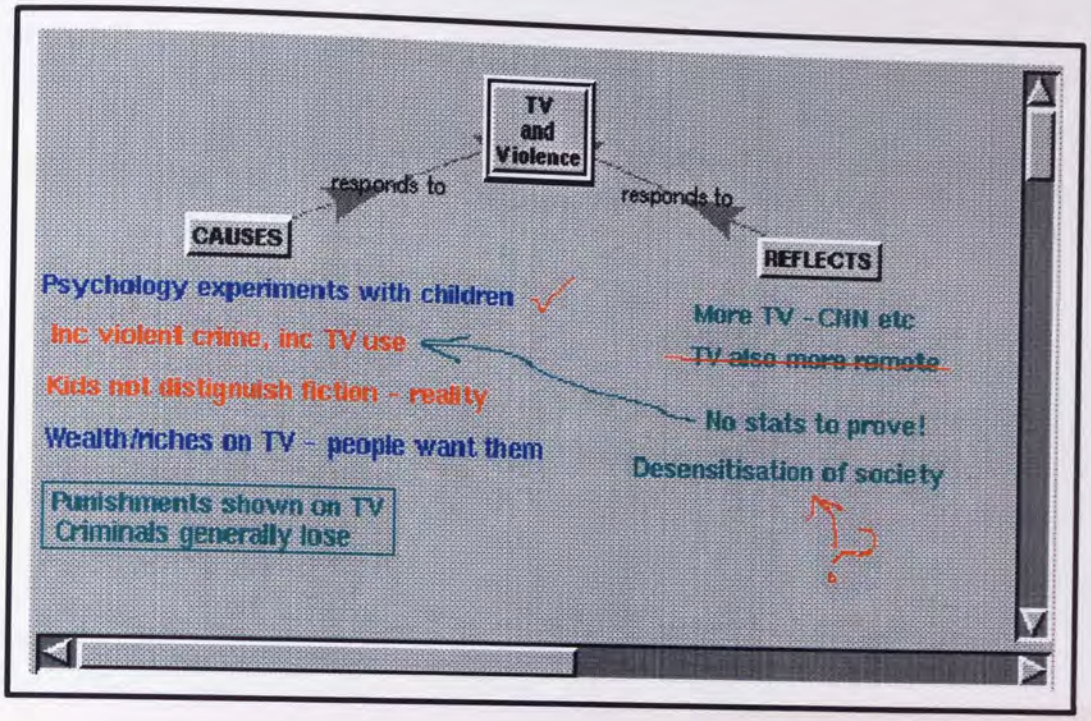

Figure 57: An unstructured argumentation session reverting to a brainstorming session.

On discussing the trial with the group, the members found the unstructured aspects of the tool more expressive in that they could write what they wished without phrasing their text for specific purposes (such as for an issue, position or argument). They also felt that they could emphasise points better using the freehand tool for underlining or placing asterisks beside a particular item.

Comparing both methods of argumentation, the group said that, while they enjoyed using the unstructured tool more than the IBIS system, the results obtained from the IBIS system were more meaningful for longer–term use. Looking at the unstructured results they could identify who had written particular items by the colour used but could not see how many people agreed with the item. For the short–term this did not matter but, at a later date, may have been important. The IBIS method forced the members to think in a specific structured manner

which, while difficult and sometimes frustrating, resulted in the participants being satisfied with the final outcome.

7.1.3 Collaborwriter

One trial was performed using a mix of the Collaborwriter session manager, authoring and negotiation tools. Five people were involved in the trial but, due to resource constraints, only three users could work simultaneously at any one time. This trial was aimed at examining the access control mechanisms in the session manager and the link between the authoring and negotiation elements of the system.

The group was asked to write a report on their views of the computing infrastructure at their work, which had been in place for the last ten years and was due for a major change during the following six months. The report should address their views and fears of the capabilities the new system would offer. The group consisted of four people from three divisions in the organisation and one person from the division implementing the change.

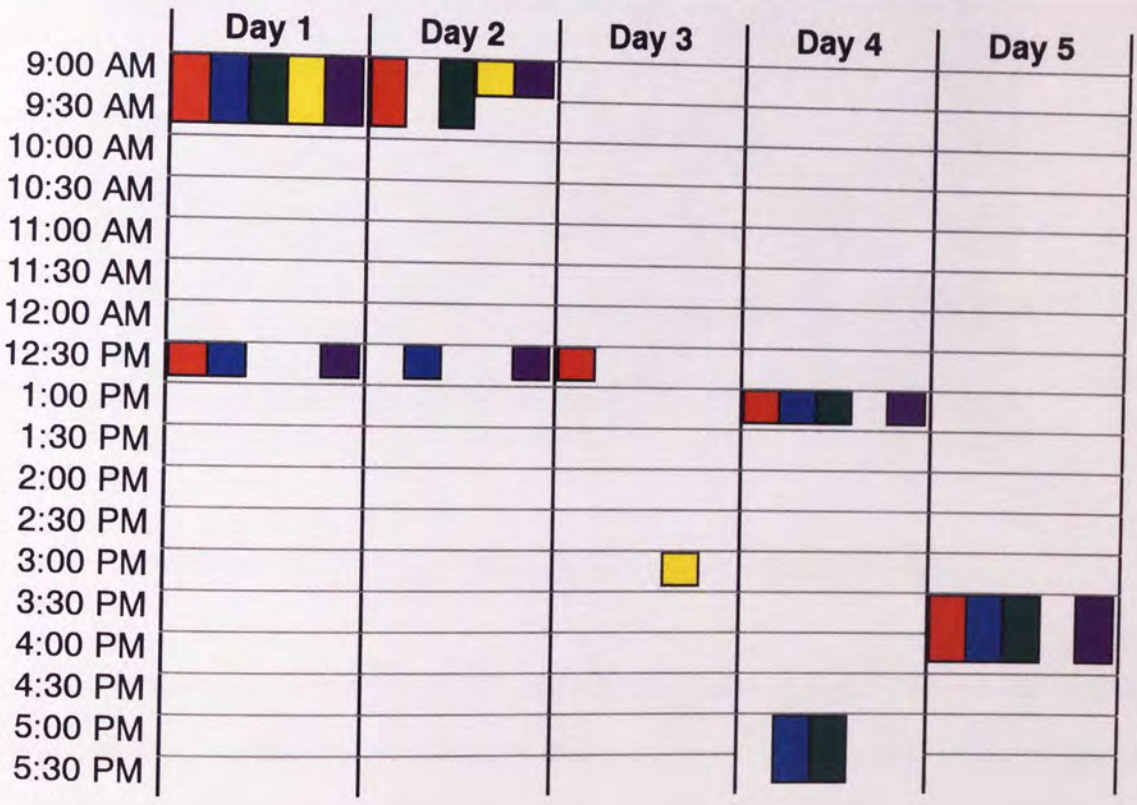

Table 12: Pictorial representation of the time and frequency for five people working on a writing project throughout five days.

For this test the system was running for five days but, as this was being performed in a working environment, limited time was available by each of the members for participating in the trial. One of the members actually had to leave the trial after Day 3, being called away on other work. Table 12 above shows the time and frequency of the participation by the five members.

In a similar manner to the authoring group, the members initially decided what they felt the main issues were, assigning areas of work to different members. In contrast with the authoring group, however, the work was not divided into segments for individuals but into segments for one, two or three members. The members also gave each other full rights to work on one another's nodes, so the access right mechanism available in the system was not used in a restrictive manner.

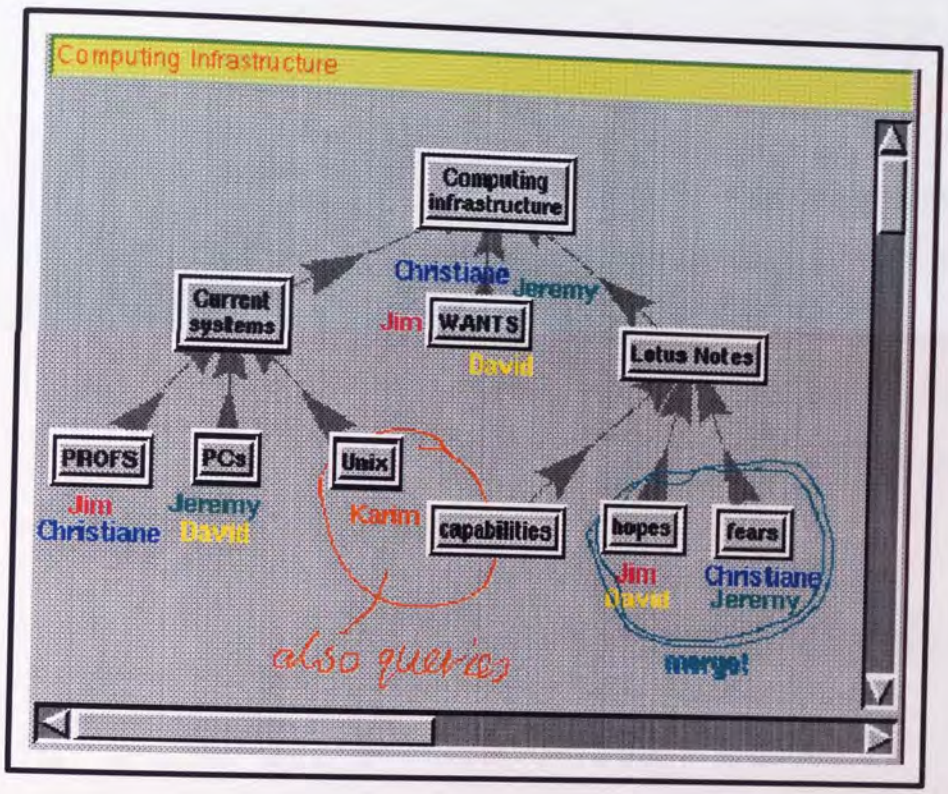

Figure 58: The negotiation tool used to coordinate activities.

The group had a choice of splitting the work between hierarchical projects at the session manager level or creating hierarchical nodes at the authoring tool level. The members chose to use the latter, feeling that this represented a "single document" for shared working more than separate projects. They also felt that the project/sub–project divisions were more for separate groups working in isolation from one another but towards a common goal. That is, they each

needed to be aware what the other was doing but did not need to be able to influence it in any way. On creating the initial nodes, the participants would have liked to be able to name the nodes individually as did the authoring group. As a work–around the members decided to use the negotiation tool in an unstructured manner for the top node to map out what the other nodes would contain, along with which members would do the work in each node. As shown in figure 58 above, the negotiation tool was thus used to formalise what work was required and who would do it.

When using the system the members rarely worked in isolation as only the mornings, lunch period and evening were available for the people to participate in the trial. When starting a session all the participants scanned what had been done previously. Two of the group also opened the negotiation tool at this stage to see if changes had been made to the central node structure.

Similar concerns were raised as with the authoring group regarding knowing what had changed, especially where a comment had changed, as this was not represented in the system. Whilst the system flagged where there was a comment, it did not show if it had been updated since the participant's last entry to the node. Members subsequently had a tendency to re–check either all or none of the comments in their node, missing on one occasion where an author had replaced a previous commentary with new text.

User [redacted] represented the user who was involved with the implementation of the new system. He provided most of the comments in the system, pointing out where the new system was an improvement on the old and justifying some of the decisions which had been made. He also directed the other members to view a pattern note he had created in the negotiation tool depicting a schematic for the new infrastructure. It was hoped that some argumentation would occur between this user and the rest as two of the group had strong feelings for the old system. Unfortunately such an argumentation did not occur.

On discussing the system the users felt that the division between projects, teams and people would be more relevant when scaling the system up to an organisational level where the session manager would represent all projects being conducted by all groups. In this case the

access controls would be of paramount importance. The split between project nodes and authoring/negotiation tools were also seen as effective, although they questioned the need for sub–projects at the session manager level.

The comments on the authoring tool were similar to that of the authoring group, mainly concerned with the import and export of text and the formatting controls available in the tool. Furthermore, one user expressed concern about the use of colour as each participant effectively used four colours – one colour representing the user, a lighter variant for locking, one lighter still for selections and a darker variant for deletions. He wondered how many users would be able to use the system without the colours between users becoming indistinguishable.

Little was said about the negotiation tool as little argumentation occurred and no voting on any particular issues took place. Reference was made more to the shared whiteboard aspects of the tool, which were felt to be effective and useful as a formal "shared memory" to the system. The users were unable to comment on the value of structured versus unstructured argumentation methodologies.

7.1.4 Deficiencies of the Tests

The preliminary user trials for Collaborwriter have taken place informally within an industry setting. Subsequently the participants were unable to spend long amounts of time with the system and thus only performed cursory trials. Some aspects of the system have therefore not been fully tested. These concern the session manager protocols for adding and removing project members as no people were added or removed from any of the projects. The blend between authoring and negotiation was not investigated fully as it was not used by the participants, nor has a wider use of the tool using multiple project nodes been examined.

In addition, as the group members all knew each other and the tasks undertaken did not have any explicit end value for the members, (the text was not used and no grades were given to it), all the groups chose to adopt an informal access right policy with all members having equal rights in all nodes. Subsequently the value of the inheritance mechanisms or the read, write, comment, delegate access policies could not be determined within the trials.

7.2 Summary of Objectives

As outlined in section 1.2 the main objective of this dissertation was to develop a collaborative writing tool. In answer to this objective Collaborwriter has been developed, taking into account the group processes and writing issues identified in chapters two and three. Collaborwriter's features include:

- A flexible project management interface with an inheritance–based role mechanism allowing both positive and negative access rights which can aid coordination and help prevent structural conflicts;
- Authoring and negotiation elements based on the writing model;
- Structured and unstructured argumentation tools to help coordinate activities and reduce conflicts;
- A text element supporting both synchronous and asynchronous writing;
- A synchronous/asynchronous voting tool with flexible voting styles; and
- "Environmentally–aware" user–interface widgets to aid author awareness.

Preliminary user trials have been conducted which show that collaborative writing is supportable and that using a system like Collaborwriter can lead to gains in areas such as a structured coordination of roles and resources, improved author awareness, reduced conflict and improved conflict resolution, and reduced redundancy of work.

7.3 Conclusions

A number of conclusions can be drawn regarding collaborative working in general and collaborative writing in particular as a result of the development of Collaborwriter and through the preliminary user trials.

7.3.1 Collaborative writing is complex but supportable

The main conclusion that can be reached is that collaborative writing is a complex but supportable task. This complexity arises at both technical and social levels.

From a social viewpoint users have many ways of working. This can range from a single user having different methods for performing different tasks to a group of users having different methods for performing a similar task. In addition, people have different ways of working together and, depending on the task, the way they want to work together can vary. For example, an author may wish to work in isolation when writing a document but then work in close collaboration with others when editing it.

When working closely together, authors may choose different forms of collaboration as suggested by the methods for floor control. During the user trials social protocols played a major part in the collaboration process, with a "free–floor" policy being used (with no locking of the text) but the participants informally acknowledging who "owned" the text. The members did not edit another's text during the writing stage but would only work on it when the group members were working together to consolidate the final document.

Many new protocols will be required for groupware, enforced either socially or formally in the system. This is inevitable with any new technology which, whilst being designed to adapt to the users' needs, will also result in the users forming new ways of working. For example, the access rights of new users in the system will depend on the type of group. A formal group may give newcomers only read access to all nodes and write access to specific nodes while an informal group might initially give full rights to all nodes. In the first case, protocols need to be established to decide how long a person is a newcomer and ensure that one person with delegate rights does not delegate full access rights for all nodes to other newcomers. In the second case, protocols need to be established concerning the "ground rules" of what one can and cannot do to another's text. Also, as with any group, newcomers need to learn a range of group norms and rituals which may not be apparent when working in a removed environment.

From a technical viewpoint the problems relate to the need for adaptable dialogs and programs, changing not only on the request of a local user but also as a result of remote users' actions. Dialogs not only need to be *modeless* but also *userless*. Users can have orientation problems with the system changing "behind their back" so userless dialogs enable any change made by any user to appear instantaneously on–screen. Collaborwriter implements userless dialogs by making them environmentally–aware. Thus the dialogs will pick up user events

and changes and, if relevant, deal with them inside the dialog. This could involve changing a list of users in a node properties... dialog or adding/removing a user's ability to perform some action in the dialog if their right to perform that action has been added or removed.

In addition, database solutions relating to multiple access of a same node work only if very fine–grained locks are permitted – a user cannot lock a complete document in a collaborative environment unless the social protocols present within the environment permit such an action. Collaborwriter allows locking to be either coarse or fine–grained, ranging from a complete node to a single character. While this lock prevents a user from explicitly changing the text, that user may still read and comment on the text if he has the appropriate access rights.

Furthermore, "race conditions" and timing problems plague groupware applications. The general solution is to route all calls through a single node which subsequently results in performance problems as that node gets overloaded with network traffic. Other possible solutions include implementing a callback handler and messaging system protocol based on request–reply messages or creating "wait loops" where the system waits for the network to supply its required information. Collaborwriter uses a blend of all three methodologies where required. For example, the multitext widget uses serialised calls through a unique node to ensure the integrity of any text created. Alternatively the interface between session manager and authoring and negotiation tools is implemented using a message request–reply protocol as communications between the two systems occur after very specific events (such as a user being added or removed from the system). Finally wait states are used by the IbisIcon widget to find out the colour of a particular user in order to implement the choice button functionality. This is needed as a new entrant receives a large amount of information from the server, and has often parsed much of the information (such as widget creation) before all the information it needs to create the widget is available.

7.3.2 Awareness is vital within a collaborative endeavour

Effective collaboration occurs best when the group members know what the other participants are doing, what their specialities are, and what they expect from you. This can be

summarised by group awareness, the key element to collaborations, and which has two main aspects.

The first awareness aspect refers to what somebody is currently doing. This can be achieved through user–interface controls at different levels of granularity. For example, a person may simply wish to know that another is working in 'Chapter 3' while another may want to know whether that person is working on some specific element described in chapter 3. Collaborwriter handles these awareness aspects through the NodeIcon and text node widget sets, either showing which node someone is working on or 'following the user' inside a particular node.

The second awareness aspect is concerned with what others can do. For example, a user can see what another user is generally able to do from the inherited node access right mechanism. Alternatively he can see if the user has any direct abilities in the current node through looking if they have written or commented in it previously from the colour–coding in the multitext widget, or are actually present in the node from the picturebar and group scrollbar widgets. By knowing who else can impact on your work, one can ensure that they know what you want to do. This could be through using the negotiation tool to define informal "boundaries of responsibility" or by sub–dividing the authoring nodes further into more specific units with appropriate access rights.

In both cases, the vital element to awareness is *communication*, be it by the system with regard to users' current position or by the users through electronic mail, the shared whiteboard, or some other means.

7.3.3 Collaborative tools cannot dictate policies but must adapt to needs

As mentioned previously people have many styles and ways of working. A collaborative tool cannot impose a way of working on a group but only suggest methods and adapt to what the users require.

Collaborwriter uses an open policy by default, letting the users add restrictions to the system as required. A free–floor floor control policy is also adopted, with users able to influence this

through the locking mechanism and by visually seeing what affect their actions have on others through the awareness mechanism.

7.3.4 Conflict and negotiation is part of collaboration

Negotiation can be seen as a key element of any collaborative endeavour. Where different people work together, so they will bring different views and positions to various projects and problems. Negotiation is the tool that can be used by different users for communicating their respective positions with a view of reaching a consensus on how to proceed.

In the same manner, however, conflict is inevitable. A conflict may be open, where members blatantly disagree with one another and need to work through the problem to try and find a solution, or hidden, where the problems are not stated and are only brought forward at a much later date. The first form of conflict is a healthy form that may even improve the quality of the end product. The second form of conflict can be very destructive, however, and is often merged with other conflicts to finally form one "super–conflict" which can have devastating effects on a project. Through tools that increase awareness, Collaborwriter attempts to reduce the potential of destructive conflicts by, for example, informing users what each are doing so that problems can be focused on and dealt with during an early stage of the project.

7.3.5 Structured argumentation networks are difficult to create

From the preliminary trials it has been observed that people have trouble formulating structured arguments. Generally this appears to be the result of the wording used for specific issues, positions or arguments, thus leaving people unsure where they should link their own statement to.

Having an enforced IBIS methodology was also found to be frustrating to users in some circumstances. Once the argumentation network was formed, however, the users found it easier to relate to and understand and saw it as an effective tool for portraying the various issues, positions and arguments of the other members in a group.

7.3.6 Environmentally–aware software components are a means of abstracting collaboration

Collaborative applications are difficult to develop. In addition to powerful toolkits such as GroupKit, environmentally–aware software components are a useful means of abstracting collaboration. The component thus becomes a self–contained element, being capable of doing some task for one user or adapt it for use with multiple users.

The multitext widget used in Collaborwriter is an example of an environmentally–aware software component. When one user is present the widget acts like a normal text editor with revision management capabilities. When others join the node, however, the widget adapts by including remote cursors, highlighters and locks for each new entrant. In addition, the widget sends details of itself to the new entrant for updating their data structures and screen display. When a person leaves, the widget again adapts by removing the relevant cursors and highlights from all the remaining users' displays.

7.4 Further Directions for Research

When starting this dissertation the use of computers to aid collaborative working and communication were still in its infancy and the World Wide Web did not exist. As a result of low–cost, high–performance networks and systems collaborative tools and groupware applications are now growing rapidly in use.

Collaborwriter has been based on the writing architecture of SGML. This architecture is now in world–wide use through its application in the development of the World Wide Web, a global network which has enabled people to become more focused on communicating and working together. While tools are appearing commercially to support users working together, much work still needs to be done.

With regard to collaborative authoring, no commercial tool exists to seamlessly support working in both asynchronous and synchronous environments. Collaborwriter provides an example of one implementation for such a system. The system needs further work on the inclusion of actual authoring tools (a text formatter and printer, spelling checker, and

thesaurus) and the ability to import and export to other file formats. In addition, the system needs to be run under more intensive trials for validation of the writing model presented in chapter four. From the initial tests it was observed that the central elements of the model were followed (coordination, writing and annotation, negotiation, consolidation) although the iterative nature of the process varied between members in a project. Further work needs to be carried out to identify how authoring systems can aid authors more in their task.

Finally, a large amount of research is required on how the World Wide Web can be used as a medium for collaboration. To date the Web has been based on asynchronous use, a situation which is currently changing with the adoption of more powerful browsing tools and scripting languages. Similarly, the Web could be seen as an ideal medium for communication, with an expressive yet structured writing language and semi–replicated architecture. Work should be carried out to see how the browser technology can be improved to display synchronous collaborations with an effective response time and in tools and protocols to help developers write groupware applications for the Web.

If the challenges presented by the World Wide Web are met, and developers can design software to enhance the ability for groups to work together when geographically dispersed, so the future of remote working patterns and a global market look more promising.

List of References

Adobe Systems Inc. (1990) *PostScript Language Reference Manual (2nd Edition)*. Reading, MA: Addison–Wesley.

Adobe Systems Inc. (1993) *Portable Document Format Reference Manual*. Reading, MA: Addison–Wesley.

Ahuja, S.R., Ensor, J.R. & Horn, D.N. (1988) The Rapport multimedia conferencing system. In *Proceedings of the Conference on Office Information Systems* (Palo Alto, California): 1–8.

Asch, S.E. (1955) Opinions and social pressure. *Scientific American* 193: 31–55.

Back, K.W. (1951) Influence through social communication. *Journal of Abnormal and Social Psychology* 46: 9–23.

Bales, R.F. (1953) The equilibrium problem in small groups. In *Working Papers in the Theory of Action* pp.111–161. New York, NY: The Free Press.

Bales, R.F. (1954) In conference. *Harvard Business Review* 33: 44–50.

Bales, R.F. & Borgatta, E.F. (1955) Size of group as a factor in the interaction profile. In *Small Groups: Studies in Social Interaction* pp.396–413. New York, NY: Knopf.

Baron, R.S., Kerr, N. & Miller, N. (1992) *Group Process, Group Decision, Group Action*. Buckingham, UK: Open University Press.

Bartlett, C.A. & Ghoshal, S. (1992) What is a Global Manager? *Harvard Business Review* Sept–Oct, pp.124–132.

Baxter, L.A. (1982) Conflict management: an episodic approach. *Small Group Behavior* 13(1): 23–42.

Beach, L.R. & Mitchell, T.R. (1990) Image theory: A behavioral theory of image making in organizations. In B. Staw and L.L. Cummings (Eds.) *Research in organizational behavior* (Vol. 12 pp.1–41). Greenwich, CT: JAI Press.

Berners–Lee, T. & Connolly, D. (1993) *Hypertext Markup Language* (Internet Draft). The World Wide Web Consortium: www.w3.org

Bottger, P.C. & Yetton, P.W. (1987) Improving group performance by training in individual problem solving. *Journal of Applied Psychology* 72: 651–657.

Brooks, F.P. (1975) *The Mythical Man–Month: Essays on Software Engineering*. Reading, MA: Addison–Wesley.

Bryan, M. (1989) *SGML: An author's guide to the Standard Generalized Markup Language.* Wokingham, UK: Addison–Wesley.

Buzan, T. (1983) *Use both sides of your brain.* New York, NY: EP Duton.

Carment, D.W. (1961) Ascendant–submissive behaviour in pairs of human subjects as a function of their emotional responsiveness and opinion strength. *Canadian Journal of Psychology* 15: 45–51.

Church, R.M. (1962) The effects of competition on reaction time and palmar skin conductance. *Journal of Abnormal and Social Psychology* 65: 32–40.

Clark, H.H. (1977) Inferences in comprehension. In D. LaBerge and S.J. Samuels (Eds.), *Basic processes in reading: Perception and comprehension.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Cockburn, A. & Greenberg, S. (1993) Making Contact: Getting the Group Communicating with Groupware. In *Proceedings of the Conference on Organizational Computing Systems* (Milpitas, California).

Coke, E.U. (1982) Computer Aids for Writing Text. In D.H. Jonassen (Ed.), *The Technology of Text: Principles for Structuring, Designing, and Displaying Text.* Englewood Cliffs, NJ: Educational Technology Publications.

Collaros, P.A. & Anderson, L.R. (1969) Effect of perceived expertness upon creativity of members of brainstorming groups. *Journal of Applied Psychology* 53(2): 159–163.

Conklin, J. & Begeman, M.L. (1988) gIBIS: A hypertext tool for exploratory policy discussion. In *Proceedings of the Conference on Computer Supported Cooperative Work* (Portland, Oregon): 140–152.

Coombs, J.H., Renear, A.H. & DeRose, S.J. (1987) Markup systems and the future of scholarly text processing. *Communications of the ACM* 30 (11): 933–947.

Cowan, D.A. (1986) Developing a process model of problem recognition. *Academy of Management Review* 11: 763–776.

Crott, H.W., Kayser, E. & Lamm, H. (1980) The effects of information exchange and communication in an asymmetrical negotiation situation. *European Journal of Social Psychology* 10: 149–163.

Dahrendorf, R. (1959) *Class and Class Conflict in Industrial Society.* London, UK: Routledge and Kegan Paul.

Dalkey, N. (1969) *The Delphi method: An experimental study of group decisions.* Santa Monica,CA: The Rand Corporation.

Darley, J.M. (1966) Fear and social comparison as determinants of conformity behavior. *Journal of Personality and Social Psychology* 4: 73–78.

DeBono, E. (1985) *Conflicts: A Better Way to Resolve Them.* Harmondsworth, UK: Penguin.

Deutsch, M. (1949) An experimental study of the effects of cooperation and competition upon group process. *Human Relations* 2: 199–231.

Deutsch, M. (1969) Conflicts: productive and destructive. *Journal of Social Issues* 25(1): 7–41.

Deutsch, M. & Gerard, H.B. (1955) A study of normative and informational social influence upon individual judgement. *Journal of Abnormal and Social Psychology* 51: 629–636.

Dourish, P. & Bellotti, V. (1992) Awareness and Coordination in Shared Workspaces. In *Proceedings of the Conference on Computer Supported Cooperative Work* (Toronto, Canada): 107–114.

Dubé–Rioux, L. & Russo, J.R. (1988) An availability bias in professional judgement. *Journal of Behavioral Decision Making* 1: 223–237.

Duchastel, P.C. (1979) Learning objectives and the organization of prose. *Journal of Educational Psychology* 71: 100–106.

Duchastel, P.C. (1982) Textual Display Techniques. In D.H. Jonassen (Ed.), *The Technology of Text: Principles for Structuring, Designing, and Displaying Text.* Englewood Cliffs, NJ: Educational Technology Publications.

Dyson, J.W., Godwin, P.H. & Hazlewood, L.A. (1976) Group composition, leadership orientation, and decision outcomes. *Small Group Behavior* 7(1): 114–128.

Easterbrook, S.M., Beck, E.E., Goodlet, J.S., Plowman, L. Sharples, M. & Wood, C.C. (1992) A Survey of Empirical Studies of Conflict. In S.M. Easterbrook (Ed.), *CSCW: Cooperation or Conflict?* London: Springer–Verlag.

Falk, G. (1981) Unanimity versus majority rule in problem–solving groups: a challenge to the superiority of unanimity. *Small Group Behavior* 12(4): 379–399.

Farallon (1988) *Timbuktu user's guide.* Farallon Computing Inc., Berkeley, California.

Feiner, S. (1988) Seeing the Forest for the Trees: Hierarchical Display of Hypertext Structure. In *Proceedings of the Conference on Office Information Systems* (Palo Alto, California): 205–212.

Feldman, D.C. (1984) The development and enforcement of group norms. *Academy of Management Review* 9: 47–53.

Fields, A. (1982) Getting Started: Pattern Notes and Perspectives. In D.H. Jonassen (Ed.), *The Technology of Text: Principles for Structuring, Designing, and Displaying Text.* Englewood Cliffs, NJ: Educational Technology Publications.

Fish, R.S., Kraut, R.S. & Leland, M.D.P. (1988) Quilt: A collaborative tool for cooperative writing. In *Proceedings of the Conference on Office Information Systems* (Palo Alto, California): 30–37.

Fisher, C.D. & Gitelson, R. (1983) A meta-analysis of the correlates of role conflict and ambiguity. *Journal of Applied Psychology* 68: 320–333.

Flower, L.S. & Hayes, J.R. (1980) The dynamics of composing: Making plans and juggling constraints. In L.W. Gregg & E.R. Steinberg (Eds.), *Cognitive Processes in Writing.* Hillsdale, NJ: Lawrence Erlbaum.

Flower, L.S. & Hayes, J.R. (1981) A cognitive theory process of writing. *College Composition and Communication* 34, 4: 365–387.

Forsyth, D.R. (1983) *An introduction to group dynamics.* Monterey, CA: Brooks/Cole Publishing Company.

Franken, R.E. & Rowland, G.L. (1979) Nature and the representation for picture-recognition memory. *Perceptual and Motor Skills* 49: 619–629.

Garland, H. & Newport, S. (1991) Effects of absolute and relative sunk costs on the decision to persist with a course of action. *Organizational Behavior and Human Decision Processes* 48: 55–69.

Gero, A. (1985) Conflict avoidance in consensual decision processes. *Small Group Behavior* 16(4): 487–499.

Gersick, C.J.G. (1988) Time and transition in work teams: Toward a new model of group development. *Academy of Management Journal* 31: 9–41.

Goldfarb, C.F., Mosher, E.J. & Peterson, T.I. (1970) An online system for integrated text processing. *Proceedings of the American Society for Information Science* 7: 147–150.

Goldfarb, C.F. (1990) *The SGML Handbook.* Oxford, England: Clarendon Press.

Greenberg, S. (1991) Personalizable groupware: Accommodating individual roles and group differences. In *Proceedings of the European Conference on Computer Supported Cooperative Work* (Amsterdam, Holland): 17–32.

Greenberg, J. & Baron, R.A. (1993) *Behavior in Organizations: Understanding and Managing the Human Side of Work (4th Edition).* Needham Heights, MA: Allyn and Bacon.

Grudin, J. (1989) Why Groupware Applications Fail: Problems in Design and Evaluation. *Office: Technology and People* 4, 3: 245–264.

Gustafson, D.H., Shulka, R.K., Delbecq, A. & Walster, W.G. (1973) A comparative study of differences in subjective likelihood estimates made by individuals, interacting groups, Delphi groups, and nominal groups. *Organizational Behavior and Human Performance* 9: 280–291.

Guy, V. & Mattock, J. (1991) *The New International Manager: An Action Guide for Cross-Cultural Business.* London, UK: Kogan Page.

Haake, J.M. & Wilson, B. (1992) Supporting Collaborative Writing of Hyperdocuments in SEPIA. In *Proceedings of the Conference on Computer Supported Cooperative Work* (Toronto, Canada): 138–146.

Haber, R.N. (1970) How we remember what we see. *Scientific American* 222(5): 104–122.

Hackman, J.R. (1991) Group influences on individuals in organisations. In M.D. Dunnette (Ed.), *Handbook of industrial/organizational psychology (2nd Edition)*. Palo Alto, CA: Consulting Psychologists Press.

Halasz, F. (1988) Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *Communications of the ACM* 31, 7: 836–852.

Hall, J. (1971) Decisions, decisions, decisions. *Psychology Today* 5: 51–54, 86–87.

Halliday, M.A.K. (1980) *Cohesion and register.* Paper presented before the conference of the National Council for Research in English at the annual meeting of the American Educational Research Association, Boston.

Hansen, J. & Haas, C. (1988) Reading and Writing with Computers: A Framework for Explaining Differences in Performance. *Communications of the ACM* 31, 9: 1080–1089.

Harrison, E.F. (1987) *The managerial decision making process (3rd Edition).* Boston, MA: Houghton Mifflin.

Harvey, O.J. & Consalvi, C. (1960) Status and conformity to pressures in informal groups. *Journal of Abnormal and Social Psychology* 60: 182–187.

Hayes, J.R. & Flower, L.S. (1986) Writing Research and the Writer. *American Psychologist* 41, 10: 1106–1113.

Henderson, D. & Card, S. (1986) Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window–based Graphic User Interface. *ACM Transactions on Graphics* 5, 3.

Hermann, M.G. & Kogan, N. (1977) Effects of negotiators' personalities on negotiating behaviour. In D. Druckman (Ed.) *Negotiations: Social–Psychological Perspectives.* Beverley Hills, CA: Sage.

Hirsch, E.D. Jr. (1977) *The philosophy of composition.* Chicago, IL: University of Chicago Press.

Hogg, M.A. & Turner, J.C. (1987) Social identity and conformity: A theory of referent informational influence. In W. Doise and S. Moscovici (Eds.), *Current Issues in European Social Psychology vol. 2.* Cambridge, UK: Cambridge University Press.

Holsapple, C.W., Lai, H. & Whinston, A.B. (1990) A Formal Basis for Negotiation Support System Research. *Kentucky Initiative for Knowledge Management Paper 25*, University of Kentucky.

Howell, J.P., Dorfman, P.W. & Kerr, S. (1986) Moderating variables in leadership research. *Academy of Management Review* 11: 88–102.

Huczynski, A.A. & Buchanan, D.A. (1991) *Organizational Behaviour: An introductory text (2nd Edition)*. Cambridge, UK: Prentice Hall.

Huff, F.W. & Piantianida, T.P. (1968) The effect of group size on group information transmitted. *Psychonomic Science* 11(10): 365–366.

International Standards Organisation (1986) *Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML) (ISO 8879)*. Geneva, Switzerland: ISO.

Janis, I.L. (1972) *Victims of Groupthink*. Boston, MA: Houghton Mifflin.

Janis, I.L. (1982) Counteracting the adverse effects of concurrence–seeking in policy planning groups: Theory and research perspectives. In H. Brandstatter, J.H. Davis and G. Stocker–Kreichgauer (Eds.) *Group Decision Making*. London, UK: Academic Press.

Jessup, L.M., Connolly, T. & Tansik, D.A. (1990) Toward a theory of automated group work: the de–individuating effects of anonymity. *Small Group Research* 21(3): 333–348.

Johansen, R. (1989) User approaches to computer–supported teams. In M.H. Olson (Ed.) *Technological Support for Work Group Collaborations*. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Kahneman, D. & Tversky, A. (1984) Choices, values, and frames. *American Psychologist* 39: 341–350.

Katz, D. & Kahn, R.L. (1978) *The Social Psychology of Organizations (2nd Edition)*. New York, NY: Wiley.

Kaye, A.R. (1993) Computer Networking for Development of Distance Education Courses. In M. Sharples (Ed.) *Computer Supported Collaborative Writing*. Berlin, Germany: Springer–Verlag.

Kiczales, G. (1992) Towards a New Model of Abstraction in the Engineering of Software. In *Proceedings of the International Workshop on Reflection and Meta–level Architecture* (November).

Kiesler, S., Siegel, J. & McGuire, T.W. (1984) Social psychological aspects of computer–mediated communication. *American Psychologist* 39(10): 1123–1134.

Kimberly, J.C. (1987) Instrumental and expressive structures in groups in organizational settings. *Small Group Behavior* 17(4): 395–406.

Kintsch, W. & van Dijk, T.A. (1978) Toward a model of text comprehension and production. *Psychological Review* 85: 363–394.

Knight, G.P. & Dubro, A.F. (1984) Cooperative, competitive, and individualistic social values: An individualized regression and clustering approach. *Journal of Personality and Social Psychology* 46: 98–105.

Knuth, D.E. (1984) *The TeXbook*. Reading, MA: Addison–Wesley.

Kraemer, K.L. & King, J.L. (1988) Computer–Based Systems for Cooperative Work and Group Decision Making. *ACM Computing Surveys* 20, 2: 115–146.

Krasner, G.E. & Pope, S.T. (1988) A Cookbook for Using the Model–View–Controller User Interface Paradigm in Smalltalk–80. *Journal of Object Oriented Programming* 1(3).

Kriedte, W. (1995) Reuse of Standards documents for print and on–line interactions. Personal communication.

Lamport, L. (1986) *LaTeX, A Document Preparation System*. Reading, MA: Addison–Wesley.

Leavitt, H.J. (1951) Some effects of certain communication patterns on group performance. *Journal of Abnormal and Social Psychology* 46: 38–50.

Leland, M.D.P., Fish, R.S. & Kraut, R.E. (1988) Collaborative document preparation using Quilt. In *Proceedings of the Second Conference on Computer Supported Cooperative Work* (Portland, Oregon): 401–410.

Lewis, B.N. (1974) *New methods of assessment and stronger methods of curriculum design*. Milton Keynes, England: Institute of Educational Technology, Open University.

Likert, R. (1961) *New Patterns of Management*. New York, NY: McGraw–Hill.

Linstone, H.A. (1984) *Multiple perspectives for decision making*. New York, NY: North–Holland.

Malcolm, N. & Gaines, B.R. (1991) A minimalist approach to the development of a word processor supporting group writing activities. In *Proceedings of the Conference on Organizational Computing Systems* (Atlanta, Georgia): 147–152.

March, J.G. & Simon, H.A. (1958) *Organizations*. New York, NY: Wiley.

Martin, J. (1973) *Design of man–computer dialogues*. Englewood Cliffs, NJ: Prentice–Hall.

McGrath, J.E. (1984) *Groups: Interaction and Performance*. Englewood Cliffs, NJ: Prentice–Hall.

Mead, R. (1994) *International Management: Cross Cultural Dimensions*. Oxford, UK: Blackwell.

Meyer, B.J.F. (1975) *The organization of prose and its effects on memory*. Amsterdam, The Netherlands: North–Holland.

Meyer, B.J.F. (1980) *Signaling in text*. Paper presented at the annual meeting of the American Psychological Association, Montreal.

Michaelsen, L.K., Watson, W.E. & Black, R.H. (1989) A realistic test of individual versus group consensus decision making. *Journal of Applied Psychology* 74: 834–839.

Microsoft Press (1995) *Microsoft Word Users Manual*. Microsoft Corporation.

Mitchell, T.R. & Beach, L.R. (1990) "Do I love thee? Let me count..." Toward an understanding of intuitive and automatic decision making. *Organizational Behavior and Human Decision Processes* 47: 1–20.

Myers, D.G. & Lamm, H. (1976) The group polarization phenonemon. *Psychological Bulletin* 83: 602–627.

Neuwirth, C.M., Kaufer, D.S., Chandhok, R. & Morris, J.J. (1990) Issues in the design of computer support for co–authoring and commenting. In *Proceedings of the Conference on Computer Supported Cooperative Work* (Los Angeles, California): 183–195.

Neuwirth, C.M., Chandhok, R., Kaufer, D.S., Erion, P., Morris, J. & Miller, D. (1992) Flexible Diff–ing in a Collaborative Writing System. In *Proceedings of the Conference on Computer Supported Cooperative Work* (Toronto, Canada): 147–154.

Nunamaker, J.F., Dennis, A.R., Valacich, J.S., Vogel, D.R. & George, J.F. (1991) Electronic meeting systems to support group work. *Communications of the ACM* 34, 7: 40–61.

Olson, G.M. & Olson, J.S. (1991) User–Centered Design of Collaboration Technology. *Journal of Organizational Computing* 1, 1: 61–83.

Olson, J.S., Olson, G.M., Storrøsten, M. & Carter, M. (1992) How a Group–Editor Changes the Character of a Design Meeting as well as its Outcome. In *Proceedings of the Conference on Computer Supported Cooperative Work* (Toronto, Canada): 91–98.

Ousterhout, J.K. (1994) *Tcl and the Tk Toolkit*. Reading, MA: Addison–Wesley.

Osborn, A.F. (1957) *Applied imagination*. New York, NY: Scribner.

Pace, R.C. (1990) Personalized and depersonalized conflict in small group discussions: An examination of differentiation. *Small Group Research* 21(1): 79–96.

Paivio, A. (1975) Coding distinctions and repetition effects in memory. In G. Bower (Ed.) *The psychology of learning and motivation*. New York, NY: Academic Press.

Parsons, T. (1961) An outline of the social system. In *Theories of Society* pp.30–79. New York, NY: The Free Press.

Pearce, J.A. II. & Ravlin, E.C. (1987) The design and activation of self–regulating work groups. *Human Relations* 40: 751–782.

Pood, E.A. (1980) Functions of communication: an experimental study in group conflict situations. *Small Group Behavior* 11(1): 76–87.

Putnam, L.L. & Poole, M.S. (1987) Conflict and negotiation. In L.W. Porter (Ed.) *Handbook of Organizational Communication: An Interdisciplinary Perspective.* Beverley Hills, CA: Sage.

Rabitti, F., Bertino, E., Kim, W. & Woelk, D. (1991) A model of authorization for next–generation database systems. *ACM Transactions on Database Systems* 1, 16: 88–131.

Reder, L. & Anderson, J.R. (1980) A comparison of texts and their summaries: memorial consequences. *Journal of Verbal Learning and Verbal Behavior* 19: 121–134.

Rimmershaw, R. (1992) Collaborative Writing Practices and Writing Support Technologies. *Instructional Science* 21: 15–28.

Rittel, H. & Kunz, W. (1970) Issues as elements of information systems. Working paper #131, Institut für Grundlagen der Planung I.A., University of Stuttgart.

Robbins, S.P. (1974) *Managing Organizational Conflict: A Non–Traditional Approach.* Englewood Cliffs, NJ: Prentice–Hall.

Robbins, S.P. (1989) *Organizational Behavior: Concepts, Controversies and Applications.* Englewood Cliffs, NJ: Prentice–Hall.

Rodden, T. (1991) A survey of CSCW systems. *Interacting with Computers* 3, 3: 319–353.

Roseman, M. (1993) Tcl/Tk as a Basis for Groupware. In *Proceedings of the Tcl/Tk '93 Workshop* (Berkeley, California).

Roseman, M. (1995) When is an object not an object? In *Proceedings of the Tcl/Tk '95 Workshop* (Toronto, Canada).

Roseman, M. & Greenberg, S. (1992) GroupKit: A Groupware Toolkit for Building Real–Time Conferencing Applications. In *Proceedings of the Conference on Computer Supported Cooperative Work* (Toronto, Canada): 43–50.

Roseman, M. & Greenberg, S. (1993) Building flexible groupware through open protocols. In *Proceedings of the ACM Conference on Organizational Computing Systems* (California, USA).

Roseman, M. & Greenberg, S. (1995) Building Real Time Groupware with GroupKit, A Groupware Toolkit. *Technical Report #95/560/12,* Department of Computer Science, University of Calgary, Calgary, Alberta, Canada.

Rummelhart, D.E. & Ortony, A. (1977) The representation of knowledge in memory. In R.C. Anderson, R.J. Spiro & W.E. Montague (Eds.) *Schooling and the acquisition of knowledge.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Sari, I.F. & Reigeluth, C.M. (1982) Writing and Evaluating Textbooks: Contributions from Instructional Theory. In D.H. Jonassen (Ed.), *The Technology of Text: Principles for Structuring, Designing, and Displaying Text.* Englewood Cliffs, NJ: Educational Technology Publications.

Sarin, S. & Greif, I. (1985) Computer–based real time conferencing systems. *IEEE Comput.* 18(10): 33–45.

Schellenberg, J.A. (1959) Group size as a factor in success of academic discussion groups. *Journal of Educational Sociology* 33.

Sermat, V. (1964) Cooperative behaviour in a mixed motive game. *Journal of Social Psychology* 62: 217–239.

Severinson Eklundh, K. (1992) Problems in Achieving a Global Perspective of the Text in Computer–based Writing. *Instructional Science* 21: 73–84.

Sharples, M. & Pemberton, L. (1990) Starting from the writer guidelines for the design of user–centred document processors. *Cognitive Science Research Paper 154*, University of Sussex, England.

Sharples, M. (1993) Introduction. In M. Sharples (Ed.) *Computer Supported Collaborative Writing*. Berlin, Germany: Springer–Verlag.

Sharples, M., Goodlet, J.S., Beck, E.E., Wood, C.C., Easterbrook, S.M. & Plowman, L. (1993) Research Issues in the Study of Computer Supported Collaborative Writing. In M. Sharples (Ed.) *Computer Supported Collaborative Writing*. Berlin, Germany: Springer–Verlag.

Shaw, M.E. (1978) Communication networks fourteen years later. In L. Berkowitz (Ed.) *Group processes*. New York, NY: Academic Press.

Shaw, M.E. & Harkey, B. (1976) Some effects of congruency of member characteristics and group structure upon group behavior. *Journal of Personality and Social Psychology* 34(3): 412–418.

Shen, H. & Dewan, P. (1992) Access Control for Collaborative Environments. In *Proceedings of the Conference on Computer Supported Cooperative Work* (Toronto, Canada): 147–154.

Sherif, M. (1954) Integrating field work and laboratory in small group research. *American Sociological Review* 19: 759–771.

Sherif, M. (1956) Experiments in group conflict. *Scientific American* 195: 54–58.

Sherif, M. (1961) Conformity–deviation, norms, and group relationships. In *Conformity and Deviation* pp.159–198. New York, NY: Harper.

Slater, P.E. (1958) Contrasting correlates of group size. *Sociometry* 25: 129–139.

Smith, B.C., Rowe, L.A. & Yen, S. (1993) Tcl Distributed Programming. In *Proceedings of the Tcl '93 Workshop* (Berkeley, California).

Sproull, L. & Kiesler, S. (1986) Reducing social context cues: electronic mail in organizational communication. *Management Science* 32: 1492–1512.

Staw, B.M. (1981) The escalation of commitment to a course of action. *Academy of Management Review* 6: 577–587.

Stefik, M., Bobrow, D.G., Kahn, K., Lanning, S. & Suchman, L. (1987) Beyond the chalkboard: computer support for collaboration and problem solving in meetings. *Communications of the ACM* 30(1): 32–47.

Stevenson, M.K., Busemeyer, J.R. & Naylor, J.C. (1990) *Handbook of industrial and organizational psychology* (Vol. 1, pp.283–374). Palo Alto, CA: Consulting Psychologists Press.

Stoner, J.A.F. (1961) *A comparison of individual and group decisions involving risk.* Unpublished Master's thesis, School of Industrial Management, M.I.T.

Swigart, R. (1990) A Writer's Desktop. In B. Laurel (Ed.) *The Art of Human–Computer Interface Design.* Reading, MA: Addison–Wesley Publishing.

Swingle, P. (1970) Dangerous games. In P. Swingle (Ed.) *The Structure of Conflict.* New York, NY: Academic Press.

Tang, J.C. (1991) Findings from Observational Studies of Collaborative Work. *International Journal of Man Machine Studies* 34(3): 143–160.

Thomas, K.W. (1976) Conflict and conflict management. In M.D. Dunnette (Ed.) *Handbook of industrial and organizational psychology.* Chicago, IL: Rand McNally.

Thomas, K.W. (1992) Conflict and negotiation process in organizations. In M.D. Dunnette (Ed.) *Handbook of industrial and organizational psychology (2nd Edition).* Palo Alto, CA: Consulting Psychologists Press.

Thomas, K.W. & Schmidt, W.H. (1976) A survey of managerial interests with respect to conflict. *Academy of Management Journal* 10: 315–318.

Thorndyke, P.W. (1977) Cognitive structures in comprehension and memory of narrative discourse. *Cognitive Psychology* 9: 77–110.

Tjosvold, D. (1986) *Working together to get things done.* Lexington, MA: Lexington Books.

Trigg, R., Suchman, L. & Halasz, F. (1986) Supporting collaboration in NoteCards. In *Proceedings of the Conference on Computer–Supported Cooperative Work* (Austin, Texas): 147–153.

Tuckman, B.W. (1961) Developmental sequence in small groups. *Psychological Bulletin* 63: 384–399.

Tuckman, B.W. & Jensen, M.A.C. (1977) Stages of small group development revisited. *Group and Organization Studies* 2: 419–427.

Tulving, E. & Thompson, D.M. (1973) Encoding specificity and retrieval processes in episodic memory. *Psychological Review* 80: 352–373.

Unger, R. (1990) Conflict management in group psychotherapy. *Small Group Research* 21(3): 349–359.

Vámos, M. (1991) Cooperative Communication: Computerware and Humanware. *Journal of Organizational Computing* 1(1): 115–123.

Van Herwijnen, E. (1990) *Practical SGML*. Dordrecht, The Netherlands: Kluwer Academic Publishers.

Van Zelst, R.H. (1952) Sociometrically selected work teams increase production. *Personnel Psychology* 5: 175–185.

Vertelney, H. (1990) An Environment for Collaboration. In B. Laurel (Ed.) *The Art of Human–Computer Interface Design*. Reading, MA: Addison–Wesley Publishing.

Waller, R. (1982) Using Typography to Improve Access and Understanding. In D.H. Jonassen (Ed.), *The Technology of Text: Principles for Structuring, Designing, and Displaying Text.* Englewood Cliffs, NJ: Educational Technology Publications.

Wanous, J.P. & Youtz, M.A. (1986) Solution diversity and the quality of group decisions. *Academy of Management Journal* 29: 149–159.

Wedley, W.C. & Field, R.H.G. (1984) A predecision support system. *Academy of Management Review* 9: 696–703.

Weinberg, S.B., Rovinski, S.H., Weiman, L. & Beitman, M. (1981) Common group problems: a field study. *Small Group Behavior* 12(1): 81–92.

Wilder, D.A. & Allen, V.L. (1977) Social support, extreme social support and conformity. *Representative Research in Social Psychology* 8: 33–41.

Winn, W. & Holliday, W. (1982) Design Principles for Diagrams and Charts. In D.H. Jonassen (Ed.), *The Technology of Text: Principles for Structuring, Designing, and Displaying Text.* Englewood Cliffs, NJ: Educational Technology Publications.

Winter, E.O. (1977) A clause relational approach to English texts: A study of some predictive lexical terms in written discourse. *Instructional Science* 6: 1–92.

Wolfe, M. (1991) New Architectures for WISIWYSWIWSWYS (What I See Is What You See When I Want to See What You See). In C.W. Holsapple & A.B. Whinston (Eds.), *Recent Developments in Decision Support Systems*. Berlin, Germany: Springer–Verlag.

Wood, C.J. (1989) Challenging the assumptions underlying the use of participatory decision–making strategies: a longitudinal case study. *Small Group Behavior* 20(4): 428–448.

Zniecki, F. (1939) Social groups as products of participating individuals. *American Journal of Sociology* 44: 799–812.

Bibliography

Arnold, J., Robertson, I.T. & Cooper, C.L. (1992) *Work Psychology: Understanding human behaviour in the workplace*. London, UK: Pitman Publishing.

Baron, R.S., Kerr, N. & Miller, N. (1992) *Group Process, Group Decision, Group Action*. Buckingham, UK: Open University Press.

Deaux, K. & Wrightsman, L.S. (1988) *Social Psychology (5th Edition)*. Pacific Grove, CA: Brooks/Cole Publishing Company.

Greenberg, J. & Baron, R.A. (1993) *Behavior in Organizations: Understanding and Managing the Human Side of Work (4th Edition)*. Needham Heights, MA: Allyn and Bacon.

Hare, A.P. (1976) *Handbook of Small Group Research (2nd Edition)*. New York, NY: The Free Press.

Homans, G.C. (1950) *The human group*. New York, NY: Harcourt, Brace & World.

Huczynski, A.A. & Buchanan, D.A. (1991) *Organizational Behaviour: An introductory text (2nd Edition)*. Cambridge, UK: Prentice Hall.

Jonassen, D.H. (1982) *The Technology of Text: Principles for Structuring, Designing, and Displaying Text*. Englewood Cliffs, NJ: Educational Technology Publications.

Jonassen, D.H. (1985) *The Technology of Text Volume Two: Principles for Structuring, Designing, and Displaying Text*. Englewood Cliffs, NJ: Educational Technology Publications.

Rosen, K.H., Rosinski, R.R. & Host, D.A. (1993) *Open Computing Best Unix Tips Ever*. Berkeley, CA: McGraw–Hill.

Wexelblat, A. (1992) *Building CSCW interfaces*. Tutorial at the ACM 1992 Conference on Computer–Supported Cooperative Work (Toronto, Canada). ACM Press.

Wilson, D.C. & Rosenfeld, R.H. (1990) *Managing Organizations: Text, Readings and Cases*. Maidenhead, UK: McGraw–Hill.

Appendices

The appendices list the program code used for the development of the collaborative applications outlined in this thesis.

Not all the program code used for the execution of the applications is included, only that code which has been uniquely written. Where a code module has been derived from other work it is presented as either a set of changes where the differences are small, or as the complete code with an acknowledgement of the originating source where the differences are large.

The code modules are split between the following appendices:

A **Tic–tac–toe application**

B **Collaborwriter Session Management**

C **Authoring Tool**

D **Negotiation Tool**

E **Groupware widgets**

F **Utility functions**

Appendix A
Tic–tac–toe application

A.1 FTP site

The tic–tac–toe application is available as part of the GroupKit software release, under the 'contributions' directory. GroupKit can be received via anonymous ftp or on World–Wide–Web from the University of Calgary, Canada at the following addresses:

Anonymous FTP:

`ftp.cpsc.ucalgary.ca` in `pub/projects/grouplab/software`

World–Wide–Web:

`www.cpsc.ucalgary.ca/pub/projects/groupkit`

A.2 Source Code Listing

This annex lists the source code listing for the Tic–Tac–Toe application described in chapter 5. The code has been developed on the GroupKit software environment and tested up to GroupKit version 3.2b (April 1996).

```
# Noughts and Crosses v.2
# by Keith McAlpine
# A simple game containing the basics for environmentally-aware software
# New version to run under GroupKit 3.1 properly

# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Utility routine to send a msg to all participants except for a
# list of exceptions  -  GroupKit utility extension
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
proc gk_toAllExcept { exceptions args } {
   set all [concat [users keys remote] [users local.usernum]]
   foreach i $exceptions {
      set idx [lsearch $all $i]
      catch { set all [lreplace $all $idx $idx] } irrelevantMsg
   }
   foreach i $all { gk_toUserNum $i eval $args }
}
```

```
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Basic initialisation routines
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# ----- Initialise conference application ------------------------------

gk_newenv tempenv
tempenv import data $argv
set thisconfname [tempenv data.confname]
tempenv destroy
gk_initConf $argv


# gk_initializeTelepointers
# telepointers don't work reliably (on a 1 user linux box, anyway!)

# ------- Create the game environment ---------------------------------

# The noughts and crosses world is stored in an environment variable
# called envXOWorld which is shared amongst all the conference
# participants. It contains the board data structure, and a list of what
# the various participants can do.
#
# Another environment variable holds the local user's view on the world.
# In this case, different users may have different game board sizes.

gk_newenv -bind -share envXOWorld
gk_newenv -bind envXOView

envXOView set lastX   -1
envXOView set lastY   -1

# ----- A global array for the collaborators menu --------------------

# xoMenu(gametype & turn & x & o & mediator)

# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Participant Join/Leave routines
# This code is designed so that the app. can adapt to its surroundings
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# ----- When somebody joins the conference, change our state to be
# a 1-2-3 player game ------------------------------------------------

proc TheWorldGrows { usernum } {
   # we need to ensure that a new user only gets joined once
   if { [envXOWorld players.x] == "" } { return }

   set playerList [concat [envXOWorld players.watching]                \
      [envXOWorld players.x] [envXOWorld players.o]                    \
      [envXOWorld players.mediator]]

   if { [lsearch $playerList $usernum] != -1 } { return }
   if { [envXOWorld gametype] == "single-user" } {
      # we've moved from a single-user game to a two-player game.
      ClearHighlight -1 -1
      if { [envXOWorld turn] == "X" } {
         envXOWorld set players.x $usernum
      } else {
         # if game is over, still make new player O by default
         envXOWorld set players.o $usernum
      }
```

```
        envXOWorld set players.mediator [envXOWorld players.x]
        envXOWorld set gametype two-player

    } elseif { [envXOWorld gametype] == "two-player" } {

        # the new member is the mediator for the game
        envXOWorld set players.mediator $usernum
        envXOWorld set gametype mediated

    } else {

        # all other arrivals can simply watch the game
        envXOWorld set players.watching
           [concat [envXOWorld players.watching] $usernum]                    \
    }
}

# ----- When somebody leaves the conference, get our world recalculated
# to allow for their departure ----------------------------------------

proc TheWorldShrinks { usernum } {

    # We only want this code called by ONE of the other conf. members
    # so set up a variable that will be either blank or irrelevant (eg a
    # past user) and use as a flag to determine if this recalculation
    # procedure has already been carried out elsewhere

    if { [envXOWorld deletedUser] == $usernum } { return }
    envXOWorld set deletedUser $usernum
    # Find out what kind of player the departing user was

    set whatDeparterWas [WhatAmI $usernum]
    switch $whatDeparterWas                                                   \
       x            { set otherPlayer o }                                     \
       o            { set otherPlayer x }                                     \
       watching {
                # the departer was only watching, so just needs to be
                # removed from the watcher list (he doesn't affect
                # the actual game in progress)
                set i [lsearch [envXOWorld players.watching] $usernum]
                envXOWorld set players.watching [lreplace                     \
                   [envXOWorld players.watching] $i $i]
                return
             }
    # Let's find a replacement player/mediator if necessary
    set mediatorReplacement ""
    if {[envXOWorld gametype] == "mediated" } {
       set mediatorReplacement [envXOWorld players.mediator]
    }
    set watchingReplacement [lindex [envXOWorld players.watching] 0]
    # Re-adjust our game world depending on who is available in the world
    # and who has left

    if { ($mediatorReplacement == "") && ($watchingReplacement == "") } {

       # there are no replacement mediators or watchers, so we'll need to
       # adjust ourselves to be a 'lesser' application if necessary
```

```
      if { $whatDeparterWas == "mediator" } {
         # we need to become a 2-play game, with no independent mediator
         envXOWorld set players.mediator [envXOWorld players.x]
         envXOWorld set gametype two-player
      } else {
         # we need to become a 1-play game, as the departer was a player
         envXOWorld set players.$whatDeparterWas                          \
            [envXOWorld players.$otherPlayer]
         envXOWorld set players.mediator [envXOWorld players.x]
         envXOWorld set gametype single-user
      }

   } else {
      # We've got a spare watcher, or we're in a mediated game, so the
      # game state may not change
      if { $watchingReplacement != "" } {
         # There's a spare watcher, so change them into whatever the
         # departer is and take the watcher off the watching list
         envXOWorld set players.$whatDeparterWas $watchingReplacement
         envXOWorld set players.watching                                  \
            [lrange [envXOWorld players.watching] 1 end]
      } else {
         # There's only the mediator to take the place of the
         # departing player, so the game changes into a 2-player game
         envXOWorld set players.$whatDeparterWas $mediatorReplacement
         envXOWorld set players.mediator [envXOWorld players.x]
         envXOWorld set gametype two-player
      }
   }
}
# ------- A new user needs their settings initialised ------------------

proc SortMeOut {} {

   # This procedure is called from an envReceived event when the
   # envXOWorld environment gets copied to the joining member process.
   # This is in preference to an 'updateEntrant' event, as there's no
   # other info to send apart from the environment data.

   for {set i 0} {$i < 3} {incr i} {
      for {set j 0} {$j < 3} {incr j} {
         AllowedToPlaceTile $i $j [envXOWorld board.$i.$j]
      }
   }
   # if the game is over, we need to get the winning line highlighted
   # (force variable to update)
   if { [envXOWorld turn] == "gameover" } {
      envXOWorld set gameover.winner [envXOWorld gameover.winner]
   }
   # get the menus kicked into life
   envXOWorld set players.mediator [envXOWorld players.mediator]
   envXOWorld set players.x [envXOWorld players.x]
   envXOWorld set players.o [envXOWorld players.o]
   envXOWorld set players.watching [envXOWorld players.watching]
   envXOWorld set gametype [envXOWorld gametype]
   envXOWorld set turn [envXOWorld turn]
}
```

```
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Code relating to the local view
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# ------- Given an x/y coordinate, return its bounding box ------------

proc CalcCoords { i j slack width } {
   set leftx [expr ($i * $width) + $slack + 4]
   set rightx [expr $leftx + $width - ($slack * 2) - 4]
   set topy [expr ($j * $width) + $slack + 4]
   set bottomy [expr $topy + $width - ($slack * 2) - 4]

   return "$leftx $topy $rightx $bottomy"
}

# ------- Place an X at position i,j ----------------------------------

proc PlaceX { i j colour } {
   set coords [CalcCoords $i $j 8 [envXOView boardSize]]
   eval " .board.c create line $coords -width 6 -fill $colour          \
      -tags ShapeAt$i.$j "
   .board.c create line [lindex $coords 0] [lindex $coords 3]          \
      [lindex $coords 2] [lindex $coords 1] -width 6 -fill $colour     \
       -tags ShapeAt$i.$j
}

# ------- Place an O at position i,j ----------------------------------

proc PlaceO { i j col } {
   eval " .board.c create oval [CalcCoords $i $j 8                     \
      [envXOView boardSize]] -width 6 -outline $col -tags ShapeAt$i.$j "
}

# ------- Draw an X or O, set up out view environment afterwards -------

proc DoPlaceTile { i j type colour } {
   .board.c delete "ShapeAt$i.$j"

   if { $type != "blank" } {
      eval "Place$type $i $j $colour"
      envXOView set lastX $i
      envXOView set lastY $j
   }
}

# ------- If we can go, put either an X or O at position i,j ----------

proc AllowedToPlaceTile { i j type } {
   DoPlaceTile $i $j $type black
}

# ------- See if we can do a tentative move on the board --------------

proc Highlight { i j } {
   ClearHighlight $i $j
   if { ([MyTurn [users local.usernum]] == 1) &&                       \
      ([envXOWorld board.$i.$j] == "blank") } {
      DoPlaceTile $i $j [envXOWorld turn] [userprefs color]
   }
}

# ------- Clear any previous highlights which may be on the board ------

proc ClearHighlight { i j } {
   set lastx [envXOView lastX]
   set lasty [envXOView lastY]
```

```
    if { (($lastx != $i) || ($lasty != $j)) && (($lastx != -1) &&          \
      ($lasty != -1)) && ([envXOWorld board.$lastx.$lasty] == "blank")} {
        .board.c delete "ShapeAt$lastx.$lasty"
    }
}

# ------- Make the game board ----------------------------------------

proc MakeBoard {} {
    set width [envXOView boardSize]

    catch { destroy .board } irrelevantMsg
    pack forget .label

    frame .board -wid [expr $width*3] -hei [expr $width*3] -relief groove
    canvas .board.c -width [expr $width*3] -height [expr $width*3]

    pack .menubar .board .label -side top -fill x
    pack .board.c -in .board

    for {set i 0} {$i < 3} {incr i} {
        for {set j 0} {$j < 3} {incr j} {
            eval " .board.c create rectangle [CalcCoords $i $j 0 $width]  \
              -out blue -fill cyan -width 4 -tags {place$i,$j square$i,$j}"
            .board.c bind place$i,$j <Enter> "Highlight $i $j"
            .board.c bind place$i,$j <1> "PlaceTile $i $j"
        }
    }
    bind .board.c <Leave> "ClearHighlight -1 -1"
}

# ------- Game over condition, so get the winning squares highlighted --

proc GameOver { turnLetter winningSquares } {
    if { $turnLetter == "O" } { set cfg -outline } else { set cfg -fill }
    foreach i $winningSquares {
        .board.c itemconfigure ShapeAt$i $cfg white
    }
}

# ------- Callback from mouse press.  See if we can place a tile at the
# square ------------------------------------------------------------

proc PlaceTile { i j } {
    gk_toUserNum [envXOWorld players.mediator]                            \
        RequestToPlaceTile $i $j [users local.usernum]
}

# ------- Print a message at the bottom of the display saying the state
# of the game -------------------------------------------------------

proc SetStateMsg {} {
    if { [envXOWorld turn] == "gameover" } {
        if { [envXOWorld gameover.winner] == "draw" } {
            set msg "Game Over - A Draw!"
        } else { set msg "Game Over - [envXOWorld gameover.winner] won." }
    } else {
        set whatAmI [WhatAmI [users local.usernum]]

        if { [MyTurn [users local.usernum]] == 1 } {
            if { [envXOWorld turn] == "O" } { set whatAmI o
                set gamestate "and it's my turn"
            } elseif { ($whatAmI == "x") || ($whatAmI == "o") } {
                set gamestate "and it's not my go"
            } else {
                set gamestate "so I don't play"
```

```
        }
        set msg "I'm $whatAmI, $gamestate"
    }

    .label configure -text "$msg"
}
# ------- If the game is over, pop up a dialog saying that ------------

proc GameOverMsg { msg } {
    catch { destroy .gameover } irrelevantMsg

    toplevel .gameover
    message .gameover.msg -width 5c -justify center -relief raised -bd 2\
        -text $msg -font -Adobe-Helvetica-Medium-R-Normal--*-180-*
    button .gameover.dismiss -text "Dismiss" -command {destroy .gameover}
    button .gameover.restart -text "New Game" -command {
        NewGame
        gk_toAll catch { destroy .gameover } irrelevantMsg
    }
    # doesn't work: gk_specializeWidgetTreeTelepointer .gameover

    pack .gameover.msg -side top
    pack .gameover.dismiss .gameover.restart -side left -padx 2m -pady 2m
}
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Mediator code
# It is possible to factor off this portion of the code to a separate
# application, so that the game is unable to be played unless'somebody'
# plays as the mediator.  The mediator knows the rules, and knows
# when somebody has won the game
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# ------- Code that sees if a person can place a tile and, if they can,
# places it and checks for a game over situation ----------------------

proc RequestToPlaceTile { i j requstr } {
    if {([MyTurn $requstr] == 1)&&([envXOWorld board.$i.$j] == "blank")}{
        set turnLetter [envXOWorld turn]
        envXOWorld set board.$i.$j $turnLetter
        # find out if the game is over.  If it is, the apps will be told
        # through the environment
        if { [CheckGameOver $turnLetter $requstr] == 0 } {
            if { $turnLetter == "X" } {
                envXOWorld set turn O
            } elseif { $turnLetter == "O" } {
                envXOWorld set turn X
            }
        }
    }
}
# ------- There's only 8 possible winning combinations, so see if we've
# got any of them -----------------------------------------------------

proc CheckGameOver { player playerNum } {
    set gameDrawed 1
    foreach i {{0.0 0.1 0.2} {1.0 1.1 1.2} {2.0 2.1 2.2} {0.0 1.0 2.0}  \
        {0.1 1.1 2.1} {0.2 1.2 2.2} {0.0 1.1 2.2} {0.2 1.1 2.0}} {
        set done 1
        foreach item $i {
            if { [envXOWorld board.$item] != $player } { set done 0 }
            if { [envXOWorld board.$item] == "blank" } { set gameDrawed 0 }
```

```
        }
        # if the game is over, set up our game over condition variables
        if { $done == 1 } {
            envXOWorld set turn gameover
            envXOWorld set gameover.winningSquares $i
            envXOWorld set gameover.winner $player
            # fire a msg to loser, one to winner, and one to everybody else
            if { $player == "X" } {
                set opp [envXOWorld players.o]
            } else {
                set opp [envXOWorld players.x]
            }
            gk_toUserNum $opp GameOverMsg "Ya Boo - You Lost!!"
            gk_toUserNum $playerNum GameOverMsg "YOU WON!"
            gk_toAllExcept [list $playerNum $opp] GameOverMsg "$player won"
            return 1
        }
    }
    # if there's no more empty spaces, the game is a draw
    if { $gameDrawed == 1 } {
        envXOWorld set turn gameover
        envXOWorld set gameover.winningSquares {}
        envXOWorld set gameover.winner draw

        gk_toAll GameOverMsg "It's a Draw"
        return 1
    }

    return 0
}
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Miscellaneous routines
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# ------- Return a description of what user ?? is ----------------------

proc WhatAmI { usernum } {
    if { [envXOWorld players.x] == $usernum } { return "x"
    } elseif { [envXOWorld players.o] == $usernum } { return "o"
    } elseif { [envXOWorld players.mediator] == $usernum } {
        return "mediator"
    } else { return "watching" }
}
# ------- Return 1/0 if it is my turn or not ---------------------------

proc MyTurn { me } {
    if {(([envXOWorld turn] == "X")&&([envXOWorld players.x] == $me)) ||
        (([envXOWorld turn] == "O")&&([envXOWorld players.o] == $me))} {
          return 1
    }
    return 0
}
# ------- Initialise the environments for a new game -------------------

proc NewGame {} {
    envXOWorld set turn X

    # Blank out the board elements
    for {set i 0} {$i < 3} {incr i} {
        for {set j 0} {$j < 3} {incr j} {
```

```
            envXOWorld set board.$i.$j blank
        }
    }
}
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Bindings
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# ------- Set up our bindings for entering and leaving conferences -----

gk_bind newUserArrived      "TheWorldGrows %U"
gk_bind userDeleted         "TheWorldShrinks %U"

# ------- Set up our bindings for manipulating the view ---------------

proc EnvXOBindings { key } {
    global xoMenu

    set variable [split $key .]

    switch [lindex $variable 0]                                          \
    board {
        # the board has changed, so get our display updated for it
        AllowedToPlaceTile [lindex $variable 1] [lindex $variable 2]     \
            [envXOWorld $key]
                                                                         \
    }

    gameover {
        if { [lindex $variable 1] == "winner" } {
            # the game is over, so get the winning run highlighted
            GameOver [envXOWorld gameover.winner]                        \
                [envXOWorld gameover.winningSquares]
            SetStateMsg
        }                                                                \
    }

    boardSize {
        # resize the board as it has changed
        MakeBoard
        SortMeOut                                                        \
    }

    gametype { set xoMenu(gametype) [envXOWorld gametype] }              \

    players {
        if { [lindex $variable 1] != "watching" } {                      \
            set xoMenu([lindex $variable 1])
                [expr [envXOWorld $key] == [users local.usernum]]        \
            after 1000 "RebuildMenu
                [lindex $variable 1] [expr [envXOWorld $key]]"
        } else {
            after 1000 "RebuildMenu watching [list [envXOWorld $key]]"
        }
        SetStateMsg                                                      \
    }

    turn {
        set xoMenu(turn) [envXOWorld turn]
        SetStateMsg
    }
}

envXOWorld bind changeEnvInfo   { EnvXOBindings %K }
```

```
envXOWorld bind addEnvInfo      { EnvXOBindings %K }
envXOView  bind changeEnvInfo   { EnvXOBindings %K }
envXOView  bind addEnvInfo      { EnvXOBindings %K }

# If we are joining a conference, all the info we need will be
# automatically copied to our environment.
# We just need to get the screen display forced to update instead

envXOWorld bind envReceived     { SortMeOut }
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Basic screen display with menus
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# ------- Return the username based on the usernum id.  It's possible
# that the environment detailing it hasn't been set yet, so return an
# ERROR msg if the username does not (yet) exist ----------------------

proc MakeMsgWithUserName { premsg usernum postmsg } {
   set user [users remote.$usernum.username]
   if { $user == "" } {
      if { [users local.usernum] == $usernum } {
         set user [users local.username]
      } else {
         # the user has not been defined yet, so return an error.  This
         # can be picked up by the calling procedure and dealt with
         return "**ERROR**"
      }
   }
   return "$premsg$user$postmsg"
}
# ------- Build up half of the menu containing the game state ----------

proc RebuildMenu { what usernum } {
   set usr [MakeMsgWithUserName "" $usernum ""]
   if { ($usr == "**ERROR**") && ($what != "watching") } {
      after 2000 RebuildMenu $what $usernum
   } else {                                                          \
      switch $what
      x { .menubar.collaboration.menu entryconfigure 5 -l "$usr is X" }\
      o { .menubar.collaboration.menu entryconfigure 6 -l "$usr is O" }\
      mediator { .menubar.collaboration.menu                          \
                    entryconfigure 7 -label "$usr is Mediator" }      \
      watching {
         set numWatchers [llength $usernum]
         set msg "Watching is "
         if { $numWatchers > 1 } {
            for { set i 0 } { $i < [expr $numWatchers -1] } { incr i } {
             set msg [MakeMsgWithUserName $msg [lindex $usernum $i] ","]
             if {$msg == "**ERROR**"} {after 2000 SortMeOut; return}
            }
            set msg "$msg and "
         }
         if { $numWatchers != 0 } {                                   \
            set m2 [MakeMsgWithUserName $msg
               [lindex $usernum [expr $numWatchers -1]] ""]
            if {$m2 == "**ERROR**"} {
               after 2000 RebuildMenu $what [list $usernum]; return}
            .menubar.collaboration.menu entryconfigure 8 -label "$m2"
         } else {                                                     \
            .menubar.collaboration.menu
               entryconfigure 8 -label "Nobody else watching"
```

```
        }
      }
   }
}
# ------- Create the basic HCI display with menubar along the top ------

wm title . "$thisconfname ([users local.username])"

gk_defaultMenu .menubar
label .label -text "Noughts and Crosses"                                \
      -font -b&h-lucida-medium-i-normal-sans-*-100-*

.menubar.file.menu insert 0 cascade                                     \
      -label "Board" -menu .menubar.file.menu.width
.menubar.file.menu insert 2 command -label "New Game" -command NewGame

menu .menubar.file.menu.width
for {set i 25} {$i < 150} {incr i 25} {
   .menubar.file.menu.width add radiobutton -label "Size $I"            \
      -command "envXOView set boardSize $i"
}

.menubar.collaboration.menu configure -tearoff 0
    # don't make this a tear-off, otherwise updates wrong

.menubar.collaboration.menu add separator
.menubar.collaboration.menu add radiobutton -label "It's Xs turn"       \
   -variable xoMenu(turn) -value X -state disabled
.menubar.collaboration.menu add radiobutton -label "It's Os turn"       \
   -variable xoMenu(turn) -value O -state disabled
.menubar.collaboration.menu add separator
.menubar.collaboration.menu add checkbutton -label "X?"                 \
   -variable xoMenu(x) -state disabled
.menubar.collaboration.menu add checkbutton -label "O?"                 \
   -variable xoMenu(o) -state disabled
.menubar.collaboration.menu add checkbutton -label "Mediator?"          \
   -variable xoMenu(mediator) -state disabled
.menubar.collaboration.menu add command -l "Watching?" -state disabled
.menubar.collaboration.menu add separator
.menubar.collaboration.menu add radiobutton -label "Single-user game"   \
   -variable xoMenu(gametype) -value single-user -state disabled
.menubar.collaboration.menu add radiobutton -label "Two-player game"    \
   -variable xoMenu(gametype) -value two-player -state disabled
.menubar.collaboration.menu add radiobutton -labe "Mediated two-player"\
   -variable xoMenu(gametype) -value mediated -state disabled

# ------- Set up our help message ----------------------------------------

set help_title "About Noughts & Crosses"
set help_text {
{normal} {A simple game of tic-tac-toe for groups of 1 or more!

The game shows the basic concept of } {normalitalic} {aware} {normal} {
software in that each node knows who is present and adjusts itself
accordingly.  Each node can be:

}
{normalitalic} {Playing X, O and mediating in a 1 player game.
Playing X or O, and mediating in a 2 player game.
Playing X, O, or mediating in a mediated game.
Watching a game.

}
```

```
{normal} {As users enter and leave the world, so the roles of the other
players change.

by Keith McAlpine (mcalpink@uk.ac.aston)

}
{normalitalic} {This version also uses shared environments and views as
in GroupKit 3.1}
}

.menubar itemcommand help add command -label "$help_title"              \
   -command "gk_topicWindow .helpWindow -title {$help_title}            \
      -text {$help_text} -width 61 -height 16"

# ------- Initialise the game if we're the first in the conference -----

if [gk_amOriginator] {
      envXOWorld set players.x          [users local.usernum]
      envXOWorld set players.o          [envXOWorld players.x]
      envXOWorld set players.mediator   [envXOWorld players.x]
      envXOWorld set players.watching   {}
      envXOWorld set gametype           single-user
      envXOWorld set deletedUser        -1
      NewGame
}
RebuildMenu "watching" "[envXOWorld players.watching]"

.menubar.file.menu.width invoke 3
      # this gets the board drawn for the first time
```

Appendix B

Collaborwriter Session Management

The Collaborwriter Session Manager is used to control and manage the multi–user environment present during a collaborative session. It consists of central registrar which controls the overall inter–process communications and a number of session managers, one for each user, which control which applications are being affected by the collaboration.

B.1 Central Registrar

The central registrar consists of three shell scripts for booting up the Collaborwriter server. It is run once on a 'well–known' machine.

B.1.1 Initial calling script

```
#!/bin/sh
# ------------------------------------------------------------------------
# registrar
# Initial server code for the central registrar
#
# Based on the groupkit registrar initialisation code, but with
# additions for a collaborwriter server.

echo "Running Collaborwriter registrar..."
persist_server &
$CW_ROOT/bin/collaborwriter_server.sh &

gkwish -notk $@ <<\END_OF_SCRIPT
source $gk_library/main/registrar.tcl
END_OF_SCRIPT
```

B.1.2 Collaborwriter server initialiser

```
#!/bin/sh
# collaborwriter_server.sh - the collaborwriter server

gkwish $@ <<\END_OF_SCRIPT
source /phd/gk/cw/collaborwriter_server.tcl
END_OF_SCRIPT
```

B.1.3 Collaborwriter server

```
# ------------------------------------------------------------------------
# collaborwriter_server
# Initial server code for collaborwriter
#
# Based on the groupkit rooms initialisation code, but with additions
# for a collaborwriter server.

#
# set up the server details

gk_initApplication
gk_newenv -server collaborwriter
gk_newenv users
gk_createServer users 9000 ""

#
# create a user interface for the server, which appears only on the
# original creating machine

tk appname "Collaborwriter_Server"
wm title . "Collaborwriter"
wm resizable . 0 0

button .b -text "Quit Server" -command "cwserver_Quit"
pack .b

# ------------------------------------------------------------------------
# Quit routine to shut down the registrar
# ------------------------------------------------------------------------

proc cwserver_Quit {} {
   global env

   #
   # ??? save all the collaborwriter environment details here ...NIY

   #
   # fudge to kill the registrar processes - this is the last process
   # out of 6

   set thisProcess [pid]

   puts "Killing registrar processes..."

   for { set i [expr $thisProcess - 5] } {$i < $thisProcess} {incr i} {
      puts "killing process $i"
      exec kill -9 $i
   }

   puts "...and finally killing me!"
   destroy .
}
```

B.2 Session Manager

```
# ------------------------------------------------------------------------
#!/usr/local/bin/gkwish -f
# ------------------------------------------------------------------------

# cwreg.tcl
# Registration client for Collaborwriter
#
# v.1 November 1994
# v.2 July-August 1995, with extensions for gk3.1 / gk3.2b3
#
# This module implements a registration policy for Collaborwriter,while
# also acting as a top level tool for the creation and management of
# projects and task groups.

#
# "Conferences" in a groupkit sense are referred to as "Projects" here.
#
# Definitions:
#  A "team" is a list of people who form a group/sub-group. People can
#  be members of multiple teams (such as in business matrix structures).
#  Each person is either a "Team leader" or a "member", with the team
#  leader being able to create new sub-projects in the project. For a
#  democratic (anarchic?) system, all members can be made team leaders.
#  Teams can contain other teams, and this is allowed to form circular
#  references. There is no hierarchy in a team, a team plus its
#  subteams simply form a set.
#
#  A "project" is a container for a groupkit conference. It contains
#  nodes which can be made local or global, and inherits the features
#  of its parent node. A project can also contain links to other
#  projects, thus allowing for the sub-division of work among team
#  members.
#
#  A "project manager" is a person responsible for the management of a
#  project, deciding who can join and whether it can be deleted.
#  Project managers are delegated from the team leaders who created the
#  project.
#
#  A "desktop" is the top-level screen, and is managed by the
#  registration client. From it projects and teams can be created.
#  There is only 1 desktop for all users, and is synchronised with all
#  other desktops.
#
# The registration policy is:
#
# Who can create a team?
#  Anybody can create a team. Initially it will contain only themselves.
#
# Who can modify a team independently?
#  Only the team leader(s) for a team may modify that team, by adding
#  or removing members.
#
# How do you join a team?
#  Send a request to join team xyz email message to the team leader.
#  The team leader will decide whether you can join. This can be done
#  by dragging your picture icon onto the team you want to join.
#  ??? If multiple team leaders, can ANY decide, or must a majority or
#  unanimous decide ??
#
```

```
# How do you leave a team?
#  Send a request to leave team xyz email message to the team leader.
#  You will be automatically removed from the team. If you 'owned' any
#  nodes in any project created under that team name, you will appear
#  as a person in that project instead.
#
# Who can create projects?
#  Only team leaders can create projects. As anyone can become a team
#  leader (by creating their own team) therefore anyone can create a
#  project.
#
# Will all projects be visible to all users?
#  No. Projects inside top-level projects will only be visible to those
#  users with access to that project. Thus sensitive projects can be
#  'hidden' from other users under a general 'umbrella' project name.
#
#  ??? NOT TRUE !!! ??? Unless I change the structure from a linked
#  tree depiction to a window-based one...
#
# Who is allowed to join a project?
#  People allocated access rights to a project will only be allowed to
#  join it (as the result of a double-click on it). A project, however,
#  may have an 'automatic entry' access right (similar to
#  Anonymous/guest logins) or an 'ask' access right, where the project
#  manager(s) are asked whether the person is allowed.
#
# Who can delete a project?
#  No one person can delete a project (unless it is private to them).
#  The project managers are asked if it should be deleted. If yes, that
#  project is lost but a new project is created where the project
#  managers are ALL the authors of the project.
#  This is similar to a passing of intellectual property rights, where
#  finally one author may 'own' that part of the project they wrote.
#
# ------------------------------------------------------------------------


# ------------------------------------------------------------------------
# The 'collaborwriter' environment holds all the details of the current
# teams/members/projects defined. This environment will be shared
# amongst any cw registration clients created.
#
# layout of
#  collaborwriter:  (Note: All item types have a createFlag setting)
#     proj.<name>.members  <member list>
#                          <join yes|no|ask>
#                          <rights (yes|no|inherit)
#
#
#         NB: <member list> = [<username> <status (leader|member)>]
#                   <rights> = read yni write yni comment yni delegate yni
#    proj.<name>.tools <tool list>
#         NB: <tool list>   = <tool name> <cmd line> <icon location>
#    proj.<name>.joinedTools.<conf number>   - tool conference name
#    team.<name>.members <member list>
#    people.<name>                 - name of person in 'world'
#    desktop.icons.<widget>
#         NB: contains list of info on the widget, namely:
#               0 - icon picture (image name)
#               1 - canvas name
#               2 - label name
#               3 - icon type (proj, team, people)
#    usercolour.<username>      - list of user colours for users
#                                 in the collaborwriter world
#
```

```
# layout of localDesktop
#   desktop.nextIconPos.<type> - x,y,column pos of next icon created
# ------------------------------------------------------------------------

#
# Set up autoloading to use appropriate library routine
# (directory library MUST be a sibling directory to where this file is
# kept)

set rootDir [pwd]/[file dirname argv0]/library
set auto_path [linsert $auto_path 0 $rootDir]
unset rootDir

#
# A -f<filename> option means that users/<filename> is used instead
# of the .groupkitrc file

#
# set the registration client rolling

gk_initGenericRC $argv

#
# The -f<filename> option also means that the .cwdesktop file is read
# from: users/<filename>.desk (i.e. its not hidden!). Otherwise the
# file is read from ~/.cwdesktop.  This facility is mainly for testing
# and debugging purposes.

set gDesktopFile "~/.cwdesktop"
set gUserprefsFile "~/.groupkitrc"

foreach i $argv {
   if { [string range $i 0 1] == "-f" } {
      set name "[string range $i 2 [expr [string length $i] -1]]"
      set gDesktopFile "users/$name.desk"
      set gUserprefsFile "users/$name"
      break
   }
}

#
# special toggle so that we only create one link-binding event

set gNodeBinding 0

# ------------------------------------------------------------------------
# Set the colour preference of a particular user
# ------------------------------------------------------------------------

proc cwreg_AddUserColour { name colour } {
   collaborwriter set usercolour.$name $colour
}

# ------------------------------------------------------------------------
# Return the colour preference of a particular user
# ------------------------------------------------------------------------

proc cwreg_GetUserColour { name } {
   return [collaborwriter get usercolour.$name]
}

# ------------------------------------------------------------------------
# When you double-click on a link line (to delete it) only the original
# user receives this event, so we need to get the changes passed through
# the whole system
# ------------------------------------------------------------------------
```

```
proc cwreg_RemoveChildParentLink { nodeW parentW } {
   set node [cwreg_WidgetToStruct $nodeW]
   set parent [cwreg_WidgetToStruct $parentW]
   cwreg_RemoveChildParentLinkNode $node $parent
}

proc cwreg_RemoveChildParentLinkNode { node parent } {

   #
   # set our collaborwriter structure so that the appropriate
   # parent/children fields are removed, but only if the person is
   # allowed to break the link. People able to break the link are
   # leaders from either node - nobody else

   set stat1 [cwreg_SeeIfMemberOf $node [userprefs name] justThisNode]
   set stat2 [cwreg_SeeIfMemberOf $parent [userprefs name] justThisNode]

   cw_Debug "[userprefs name] is: $stat1 in node $node" yellow
   cw_Debug "[userprefs name] is: $stat2 in node $parent" yellow

   if { ($stat1 != "leader") && ($stat2 != "leader") } { return }

   set children [collaborwriter get $parent.children]
   set item [lsearch $children $node]
   collaborwriter set $parent.children [lreplace $children $item $item]
   collaborwriter set $node.parent ""
}

# ------------------------------------------------------------------------
# Convenience function to convert a widget path into its related
# collaborwriter structure
# ------------------------------------------------------------------------

proc cwreg_WidgetToStruct { w } {
   set type [[$w getenvironment] get icontype]
   return "$type.[$w cget -envname]"
}

# ------------------------------------------------------------------------
# Convenience function to convert a collaborwriter structure into a
# widget path
# ------------------------------------------------------------------------

proc cwreg_StructToWidget { s } {
   set s [split $s .]
   return ".[lindex $s 0].f.c.[cw_RemoveSpacesFromName [lindex $s 1] 1]"
}

# ------------------------------------------------------------------------
# Set up the desktop coordinates system so that new icons get added in
# the canvas semi-intelligently (across visible screen, then down, then
# in columns across)
# ??? There's lots of magic numbers, with the system based on the size
# (width) of the top-level node
# ------------------------------------------------------------------------

proc cwreg_NextGlobalCoords { type } {
   set coords [localDesktop get desktop.nextIconPos.$type]
   set view [".$type.f.c" xview]
   set canvasWidth [expr ([lindex $view 1] - [lindex $view 0]) *1000]
   set x [expr [lindex $coords 0] + 60]
   set y [lindex $coords 1]
   set column [lindex $coords 2]

   if { $x > $canvasWidth } {

      #
      # we're at the end of the visible canvas width
```

```
      set x 20
      set y [expr $y + 60]
      set view [".$type.f.c" yview]
      set canvasHeight [expr ([lindex $view 1] - [lindex $view 0])*1000]

      if { $y > $canvasHeight } {

         #
         # we're at the end of the visible canvas height, so goto the
         # next column
         # ??? no checks for what we do once this space is all used!!

         set y 40
         set column [expr $column + $canvasWidth + 20]
      }
   }
   localDesktop set desktop.nextIconPos.$type [list $x $y $column]
}

# ---------------------------------------------------------------------
# Get the next x,y icon coordinates for the appropriate canvas
# ---------------------------------------------------------------------

proc cwreg_GetGlobalCoords { type } {
   set coords [localDesktop get desktop.nextIconPos.$type]
   return [list [expr ([lindex $coords 0] + [lindex $coords 2])]       \
                [lindex $coords 1]]
}

# ---------------------------------------------------------------------
# A simple Yes/No dialog type, along with commands to execute on a
# YES/NO click - ??? this should be in cw_utils ???
# ---------------------------------------------------------------------

proc cwreg_YesNoDialog { msg ok cncl okcmd cnclcmd} {

   set dlog ".yesno[cw_GetCounter]"
   cw_CreateDialog $dlog
   wm title $dlog "Decision request"
   message $dlog.msg -text $msg -relief groove -bd 4 -width 6c

   $dlog.ok configure -text $ok                                        \
      -command "cw_RemoveDialog $dlog; eval \"$okcmd\""                \
   $dlog.cancel configure -text $cncl
      -command "cw_RemoveDialog $dlog; eval \"$cnclcmd\""

   pack $dlog.msg -side top
   pack $dlog.cancel -side left -padx 10
   pack $dlog.ok -side right -padx 10
}

# ---------------------------------------------------------------------
# A collaborwriter-defined event has occurred which we need to deal with
# ---------------------------------------------------------------------

proc cwreg_DealWithEvent { e data from } {

   if { ($e == "requestJoinTeam") || ($e == "requestJoinProj") } {
      cw_Debug "we've been asked to let $data join us" white
      set node [lindex $data 0]
      set newNode [lindex $data 1]
      set newMember [lindex [split $newNode .] 1]
      set theTeam [lindex [split $node .] 1]

      if { $e == "requestJoinTeam" } {
         set s [lindex $data 3]
```

```
            cwreg_YesNoDialog "$from requests \"$newMember\" to join team \
            \"$theTeam\" as a $s" Join Disallow [list                      \
            cwreg_AddDetailsToteam $node $newNode [lindex $data 2] $s] ""
         } else {
            set s [lrange $data 3 end]
            cwreg_YesNoDialog "$from requests \"$newMember\" to join       \
            project \"$theTeam\"" Join Disallow [list                      \
            cwreg_AddDetailsToproj $node $newNode [lindex $data 2] $s] ""
         }
      }
   }

# ---------------------------------------------------------------------
# Our collaborwriter environment variable has changed, so deal with it.
# ---------------------------------------------------------------------

proc cwreg_EnvironmentChanged key {
   set parts [split $key .]
   set keyItem [lindex $parts 0]
   set lastPart [lindex $parts end]

   cw_Debug "[userprefs name]:> Env changed:                              \
             $key > [collaborwriter get $key]" cyan

   if { ($keyItem == "desktop") && ([lindex $parts 1] == "icons") } {

      #
      # an icon might have changed picture on our desktop, so update it

      cwreg_GlobalDesktopChange $lastPart [collaborwriter get $key]

   } elseif { $keyItem == "cwevent" } {

      #
      # there's been an event posted, so see if it's for us

      set e [collaborwriter get $key]
      set idx [lsearch [lindex $e 3] [userprefs name]]
      if { $idx != -1 } {

         #
         # the event is for me, so deal with it. If I'm last to receive
         # it, then delete the event, otherwise remove me from the
         # recipient list. This is done so that the event still hangs
         # around until all people have received it, including those
         # currently off-line, but has the disadvantage of causing the
         # same event to be generated when the change is made.

         # ??? What we need is a mechanism for telling the server to
         # update its info, but to not bother with anyone else. When we
         # next log on, we won't get the event ourselves as it'll have
         # the correct values in the server ???

         cw_Debug "dealing with event: $e" cyan

         set to [lreplace [lindex $e 3] $idx $idx]
         if { [llength $to] == 0 } {
            collaborwriter delete $key
         } else {                                                          \
            collaborwriter set $key
               [list [lindex $e 0] [lindex $e 1] [lindex $e 2] $to]
         }
         cwreg_DealWithEvent [lindex $e 0] [lindex $e 1] [lindex $e 2]
      }
   } elseif { $lastPart == "parent" } {

      #
      # a node has a new parent - that parent's node will already have
      # its children defined
```

```
    # ??? previously we created a link and used a hierarchy for the
    # teams, with a team only being able to exist in one hierarchy.
    # Now, teams are like people, and can be placed in any other team
    # node. This can cause circular references, however,
    # (eg t1 contains t2, t2 contains t1), but is not necessarily a
    # bad thing! (eg. in the cases above, members= t1 + t2)

    # Now, only projects are hierarchically based.

    if { $keyItem == "proj" } {
       set p [cw_RemoveSpacesFromName [collaborwriter get $key] 1]
       set w [cwreg_StructToWidget $key]
       if { $p != "" } {
          $w configure -parent [cwreg_StructToWidget $p]
       } else {

          #
          # there's been a break in the links, so delete the link
          # widget. The parent info is gone in the collaborwriter
          # structure, but its still in the widget...

          $w breaklink
       }
    }
} elseif { $lastPart == "children" } {    ; # do nothing
} elseif { $lastPart == "createFlag" } {

    #
    # a new team/project/person might has been created, so get it
    # added to the approp. view (.proj / .team / .people ...initally)

    if { [lindex $parts 1] == "Anybody" } {
       set icn "anonicon"
    } else {
       set icn "[lindex $parts 0]icon"
    }
    cwreg_AddIconToView                                              \
                $keyItem [lindex $parts 1] ".$keyItem.f.c" $icn
} elseif { ($lastPart == "members") && ($keyItem != "people") } {

    #
    # the member list has changed for a node - this could have
    # ramifications on any editors running as a subset of this node.
    # So, see if there's any confs running for this node and send the
    # latest membership details to it

    cw_Debug "MEMBERS CHANGED" white
    set node [file rootname $key]
    set nodeList {}

    if { $keyItem == "team" } {

       #
       # we need to see if this team is in any of our projects

       set nodeList [cwreg_GetProjectsTeamMemberOf [lindex $parts 1]]
    } else {

       #
       # its a proj. member,so any valid nodes are this + child nodes

       set nodeList $node
    }
    if { $nodeList == "" } { return }

    #
    # we check these nodes plus any children of them (as its all
    # inherited)
```

```
      set validNodes {}
      foreach i $nodeList {
         set validNodes [concat $validNodes [cwreg_GetChildrenNodes $i]]
      }

      set confnums [confs keys c]
      foreach i $confnums {
         if { [confs c.$i.users] != "" } {
            if {[lsearch $validNodes [confs c.$i.projectNode]] != -1} {

               #
               # we've found an active conference, so send the details

               cw_Debug "updating conf node: $node confnum: $i" white
               cwreg_SendUpdatedInfoToEditor $i $node
            }
         }
      }
   }
}

# ------------------------------------------------------------------------
# Routine called when we detect a change between the global icon and
# our local icon
# ------------------------------------------------------------------------

proc cwreg_GlobalDesktopChange { widgetName winfo } {
   set w "[lindex $winfo 1].$widgetName"

   if { [info commands $w] == "" } { return }
   set newPict [lindex $winfo 0]
   set currentPict [$w cget -widget]

   cw_Debug "DesktopChange: current: $currentPict , new: $newPict" red

   # first see if we are using a custom icon (and that it's different
   # to the new one - if there's no difference then make no change)
   #
   # ??? NOTE: As the image is based on a FILENAME, this file must be
   # present IN THE SAME LOCATION as the others. For this, its best to
   # have a common file area which is shared by all users, and accessed
   # through an environment variable (e.g. use "$CWICONS/peopleicon" to
   # access the people icon)

   if { $currentPict != $newPict } {

      #
      # the picture is different, so see if it's the default one. If it
      # is, then we make the change automatically, otherwise we ask the
      # user if they want to change their custom icon to the new one

      if {($widgetName == "anybody") && ($currentPict == "anonicon")} {
         $w configure -iconPict $newPict
      } elseif { $currentPict == "[lindex $winfo 3]icon" } {

         # it's the default, so make the change automatically

         $w configure -widget $newPict
      } else {

         # there's already a different custom icon, so see if
         # the user wants to allow the change

         cw_Debug "See if change icon $currentPict to icon $newPict" red
                                                                    \
         cwreg_AskAboutIconChange
                     $w $widgetName $winfo $currentPict $newPict

      }
```

```
    }
}

# ---------------------------------------------------------------------
# If there's been a global change to an icon, and we've got a custom
# icon, ask if we want to change our custom icon into the global one
# ---------------------------------------------------------------------

proc cwreg_AskAboutIconChange {w widgetName winfo currentPict newPict} {
    set dlog ".iconChange_[cw_GetCounter]"
    set defaultCurrent [$w cget -defaulticon]
    set defaultNew "[lindex $winfo 3]icon"

    toplevel $dlog
    wm title $dlog "Change desktop icon for \"[lindex $winfo 2]\""
    frame $dlog.f1
    frame $dlog.f2
    groupIcon $dlog.f1.current -icontext "Current Icon"                    \
        -widget $currentPict -defaulticon $defaultCurrent
    groupIcon $dlog.f1.new -icontext "New Icon" -widget $newPict           \
        -defaulticon $defaultNew
    label $dlog.f1.l -text "change to >>>" -justify center -width 13
    button $dlog.f2.ok -text "OK" -width 7                                 \
        -command "$w configure -widget $newPict; cw_RemoveDialog $dlog"
    button $dlog.f2.cancel -text "Cancel" -width 7                         \
        -command "cw_RemoveDialog $dlog"

    pack $dlog.f1 $dlog.f2 -side top -pady 10
    pack $dlog.f1.current $dlog.f1.l $dlog.f1.new -side left -padx 5
    pack $dlog.f2.cancel -side left -padx 30
    pack $dlog.f2.ok -side right -padx 30
}

# ---------------------------------------------------------------------
# Return a list of the child nodes for node "node"
# ---------------------------------------------------------------------

proc cwreg_GetChildrenNodes { node } {
    set nodes [list $node]
    set children [collaborwriter get $node.children]
    foreach child $children {
        set nodes [concat $nodes $child [cwreg_GetChildrenNodes $child]]
    }

    return $nodes
}

# ---------------------------------------------------------------------
# Return the root parent node for node "node"
# ---------------------------------------------------------------------

proc cwreg_GetRootNode { node } {
    set parent [collaborwriter get $node.parent]
    if { $parent == "" } { return $node
    } else { return [cwreg_GetRootNode $parent] }
}

# ---------------------------------------------------------------------
# See if the node item "item" is in the hierarchy containing node
# "searchNode"
# ---------------------------------------------------------------------

proc cwreg_SeeIfInHierarchy { searchNode item } {
    set searchNode [cwreg_GetRootNode $searchNode]
    return [cwreg_SearchFromNode $searchNode $item]
}
```

```
# ------------------------------------------------------------------------
# Recursive routine to search the node tree for a specific item
# ------------------------------------------------------------------------

proc cwreg_SearchFromNode { searchNode item } {

   #
   # first search our current node

   if { $searchNode == $item } { return 1 }

   #
   # now check through our children

   set children [collaborwriter get $searchNode.children]
   foreach i $children {
      if { [cwreg_SearchFromNode $i $item] == 1 } { return 1 }
   }

   return 0
}

# ------------------------------------------------------------------------
# Make a node a sibling of another. The parent set needs to be done
# last as that triggers the graphical changes elsewhere
# ------------------------------------------------------------------------

proc cwreg_MakeNodeChild { parent child type } {

   if { [collaborwriter get $child.parent] == "" } {
      cw_Debug ">> Making $child a child of $parent" green
      set children [collaborwriter get $parent.children]
      lappend children $child
      collaborwriter set $parent.children $children
      collaborwriter set $child.parent $parent

      #
      # update the memberlist field as well (so that it can be read
      # from dialogs)

      set memberList [collaborwriter get $parent.members]

      # ???? NOT FULLRIGHTS, BUT INHERIT...
      # ???? lappend memberList [list $child fullrights]

      collaborwriter set $parent.members $memberList
   } else {
      cw_Debug "??? This node ($child) already has a parent ???" red
   }
}

# ------------------------------------------------------------------------
# Called as the result of a valid icon drop onto a team icon
# ------------------------------------------------------------------------

proc cwreg_AddDetailsToteam { node newNode newType status } {
   set memberList [collaborwriter get $node.members]
   foreach i $memberList {
      if {[lindex $i 0] == $newNode} {
         cw_Debug "Member already exists!" red
         return
      }
   }

   #
   # if we're the leader of the team, we add the person automatically.
   # If not, we need to request joining the team.
```

```
    # NOTE: Other people can request for others to join a team
    # If 'Anybody' is in the team, we join automatically with the same
    # privileges of him/her

    set anyBody [cwreg_SeeIfMemberOf $node "Anybody" justThisNode]
    if { $anyBody == "notFound" } {
       set automaticJoin askFirst
       set myStatus                                                    \
          [cwreg_SeeIfMemberOf $node [userprefs name] justThisNode]
       if { $myStatus == "leader" } { set automaticJoin member }
    } else {
       set automaticJoin $anyBody
    }
    if { $automaticJoin == "askFirst" } {
       cwreg_PostEvent requestJoinTeam [list $node $newNode $newType    \
          $status] [cwreg_GetLeaderList $node justThisNode]
    } else {
       lappend memberList [list $newNode $automaticJoin]
       collaborwriter set $node.members $memberList
    }
}

# ------------------------------------------------------------------------
# Called as the result of a valid icon drop onto a project icon
# ------------------------------------------------------------------------

proc cwreg_AddDetailsToproj { node newNode newType status } {
    set status [lindex $status 0]
    set memberList [collaborwriter get $node.members]
    foreach i $memberList {
       if {[lindex $i 0] == $newNode} {
          cw_Debug "Member Already exists!" red
          return
       }
    }

    #
    # if we're a project leader, we add the person automatically. If not,
    # we need to request joining the project.
    # NOTE: Other project members can request for others to join a
    # project. If 'Anybody' is present, we join automatically with the
    # same privileges of him/her

    if { ($status != "") && ([llength $status] != 1) } {
       set joinDetails $status
       if { [lindex $joinDetails 1] == "ask" } {                        \
          set joinDetails [concat [lindex $status 0] yes
             [lrange $status 2 end]]
       }
    } else {
       set anyBody [cwreg_SeeIfMemberUpwardsOf $node "Anybody"]
       set myStatus [cwreg_SeeIfMemberUpwardsOf $node [userprefs name]]

       if { [lindex $myStatus 1] == "leader" } {
          if { $anyBody == "notFound" } {                               \
             set joinDetails
               [list member yes inherit inherit inherit inherit]
          } else {                                                      \
             set joinDetails
               [concat [lindex $anyBody 1] yes [lrange $anyBody 3 end]]
          }
       } else {
          if { $anyBody == "notFound" } {                               \
             set joinDetails
               [list member ask inherit inherit inherit inherit]
```

```
            } else {
                set joinDetails [lrange $anyBody 1 end]
            }
        }
    }

    if { [lindex $joinDetails 1] == "ask" } {
        cwreg_PostEvent requestJoinProj [list $node $newNode $newType    \
            $joinDetails] [cwreg_GetLeaderList $node justThisNode]
    } else {
        lappend memberList [concat [list $newNode] $joinDetails ]
        collaborwriter set $node.members $memberList
    }
}

# ------------------------------------------------------------------------
# Recursive subroutine to form all the links in a collaborwriter
# structure. Useful when loading in, where we build the widgets and then
# need to make the links between them
# ------------------------------------------------------------------------

proc cwreg_MakeLinkHierarchy { node parent } {
    set children [collaborwriter get $node.children]
    foreach child $children {
        cwreg_MakeLinkHierarchy $child $node
    }

    if { $parent != "" } {

        #
        # quick fudge method to get the parent widget options set up

        set key "$node.parent"
        cwreg_EnvironmentChanged $key
    }
}

# ------------------------------------------------------------------------
# Set up the bindings for a widget.
# Given a list of keysyms, bind them to the current widget so that they
# are invalid entries
# ------------------------------------------------------------------------

proc cwreg_BindWidget {invalidChars widgetName} {
    set length [string length $invalidChars]
    foreach i $invalidChars {
        bind $widgetName <Key-$i> {
            cw_Debug "Invalid character\a" yellow; break}
    }
}

# ------------------------------------------------------------------------
# Create a groupIcon widget.
# Note: Tk windows must being in lowercase, which is why there is the
# string tolower function call when creating the name.
#
# Use this routine when we are updating our views (the collaborwriter
# data is already correct) e.g. on startup when we read our desktop
# file, or when we discover that other icons have been created in our
# absence.
# ------------------------------------------------------------------------

proc cwreg_CreateIcon {labelName rootCanvas iconToUse type x y} {
    global gNodeBinding

    #
    # create the icon and add it to our canvas
```

```
    set newName [cw_RemoveSpacesFromName $labelName 1]
    set w $rootCanvas.$newName

    #
    # Make sure the icon we are using exists as an image. If it doesn't,
    # first try loading it in and, if that fails, resort to the default
    # icons (procedure done in widget)

    if { $labelName == "Anybody" } {set icn "anonicon"
    } else {set icn "[set type]icon"}

    nodeIcon $w -envname $labelName -icontext $labelName                \
       -defaulticon $icn -widget $iconToUse -line arrow                 \
       -linkcolor blue -canvas $rootCanvas
    set id [$rootCanvas create window $x $y -window $w]
    [$w getenvironment] set icontype $type

    #
    # tell the icon data structure where it is on the canvas and its
    # unique canvas ID

    $w setcanvasinfo $id $x $y
    groupIcon_DragIcon $rootCanvas $w $x $y absolute

    #
    # set up our bindings for the icon:
    # Left-btn drag: - drag icon into another icon (eg person into team)
    # Mid -btn dbl-click: - modify item (open appropriate dialog)
    # Rigt-btn drag: - drag icon around on canvas

    $w bindall <Double-Button-2>                                        \
       [list cwreg_Modify$type $w $newName $labelName]
    $w bindall <1>                                                      \
       "cwreg_StartDragIconImage $w $type %X %Y;
        cwreg_StartPopupMenu $w %b $type %X %Y"
    $w bindall <B1-Motion>                                              \
       "cwreg_GetWidgetUnderMouse $w %X %Y $type;
        cwreg_RemovePopupMenu $w"
    $w bindall <B1-ButtonRelease>                                       \
       "cwreg_StopDragIconImage $w %X %Y $type;
        cwreg_RemovePopupMenu $w"
    $w bindall <2> "cwreg_StartPopupMenu $w %b $type %X %Y"
    $w bindall <B2-Motion> "cwreg_RemovePopupMenu $w"
    $w bindall <B2-ButtonRelease> "cwreg_RemovePopupMenu $w"

    #
    # if there's a break in a link, we receive a notification of it
    # so set up our handler

    if { $gNodeBinding == 0 } {                                         \
       breakLink bind nodeIconBreakLink
          "cwreg_RemoveChildParentLink %N %P"
       set gNodeBinding 1
    }

    #
    # update the global x,y coordinates

    cwreg_NextGlobalCoords $type

    return $newName
}

# ------------------------------------------------------------------
# Create a popup menu over the icon, either displaying the list of tools
# or list of members
# ------------------------------------------------------------------
```

```
proc cwreg_StartPopupMenu { w b type x y } {
   catch { destroy $w.popup }
   set node [cwreg_WidgetToStruct $w]
   if { ($b == 1) && ($type == "proj") } {
      #
      # if we arn't a member of the project, we have no rights to
      # run/join a project
      set myState [cwreg_SeeIfMemberUpwardsOf $node [userprefs name]]
      if { $myState == "notFound" } {
         set myTeams [cwreg_GetTeamsAmMemberOf [userprefs name]]
         foreach team $myTeams {
            set myState [cwreg_SeeIfMemberUpwardsOf $node $team]
            if { $myState != "notFound" } { break }
         }
         if { $myState == "notFound" } { return }
      }
      #
      # myState is changed so that it returns just the final four
      # elements to the joined conference
      # (read (yni) comment write delegate)
      set myState [lrange $myState 3 end]
      #
      # create the run/join popup menus
      menu $w.popup -tearoff 0
      menu $w.popup.run -tearoff 0
      menu $w.popup.join -tearoff 0
      $w.popup add cascade -label "Run" -menu $w.popup.run
      $w.popup add cascade -label "Join" -menu $w.popup.join

      set details [collaborwriter get $node.tools]
      foreach i $details {
         userprefs "prog.[lindex $i 0].cmd"                              \
            "exec gkwish -f [lindex $i 1]"
         $w.popup.run add command -label [lindex $i 0]                   \
            -command [list cwreg_RunToolCmd $i $node $myState]
      }

      set details [collaborwriter keys $node.joinedTools]
      foreach i $details {
         menu $w.popup.join.$i -tearoff 0

         set confname [collaborwriter get $node.joinedTools.$i]
         set userlist [confs keys c.$i.users]
         if { $userlist == "" } {
           $w.popup.join.$i add command -label "from previous time"     \
           -command [list cwreg_JoinToolCmd $confname $node $i $myState]
         }
         foreach j $userlist {                                           \
            $w.popup.join.$i add command                                 \
               -label [confs get c.$i.users.$j.username] -command        \
               [list cwreg_JoinToolCmd $confname $node $i $myState]
                                                                         \
         }
         $w.popup.join add cascade
            -label $confname -menu $w.popup.join.$i
      }
      after 200 [list cwreg_PlacePopup $type $b $w $x $y 0]
   } elseif { ($b == 2) && ($type != "people") } {
```

```
      menu $w.popup -tearoff 0
      cwreg_MakeMemberMenu $w.popup $node
      after 200 [list catch [list tk_popup $w.popup $x $y 0]]
   }
}

# ------------------------------------------------------------------------
# Show the popup menu.  If it was a button 1 drag, also remove any
# highlighting which had started with the drag event
# ------------------------------------------------------------------------

proc cwreg_PlacePopup { type b w x y item } {
   set err [catch "tk_popup $w.popup $x $y $item"]
   if { ($err == 0) && ($b == 1) } {

      #
      # the popup has happened, so get rid of the highlighting for
      # drag & drop

      cwreg_StopDragIconImage $w $x $y $type
   }
}

# ------------------------------------------------------------------------
# Remove the popup menu that's currently over the icon
# ------------------------------------------------------------------------

proc cwreg_RemovePopupMenu { w } {
   catch { destroy $w.popup }
}

# ------------------------------------------------------------------------
# Run a command from the project, called either as a menu choice or
# double-click
# ------------------------------------------------------------------------

proc cwreg_RunToolCmd { details node myDetails } {
   set cmdline [lindex $details 1]
   set filetype [file extension $cmdline]
   if { $filetype == ".tool" } {
      cw_Debug "groupware tool $cmdline to be run" green

      set id [gk_uniqprogid]
      userprefs set collaborwriter.myDetails.$id [list $node $myDetails]

      # userprefs "prog.[lindex $details 0].cmd"
      # "exec gkwish -f $cmdline"

      keylset conf confname [lindex $details 0]                    \
         conftype [lindex $details 0] originator $id projectNode $node
      gk_callNewConference $conf
      gk_pollConferences
   } else {
      set pid [exec $cmdline &]
      cw_Debug "spawned tool $cmdline as pid $pid" green
   }
}

# ------------------------------------------------------------------------
# Run a command from the project, called either as a menu choice or
# double-click
# ------------------------------------------------------------------------

proc cwreg_JoinToolCmd { tool node confnum myDetails } {
   cw_Debug "joining $tool confnum: $confnum" green
   if { $confnum != "" } {
```

```
      if { ![gk_alreadyJoined $confnum] } {
         userprefs set collaborwriter.myDetails.$confnum                \
                                        [list $node $myDetails]
         gk_conferenceJoined $confnum
         gk_callJoinConference $confnum
         gk_pollUsers $confnum
      }
   }
}

# ---------------------------------------------------------------------
# Initialise our local environment for the start of the dragging. We
# want to drag this & all child nodes of it,so get them all highlighted
# ---------------------------------------------------------------------

proc cwreg_StartDragIconImage { w type x y } {
   $w highlight yellow all
   localDesktop set dragging $w
   localDesktop set highlightWidget ""
   . configure -cursor [list @bitmaps/cursor$type black]
}

# ---------------------------------------------------------------------
# Tidy up after the dragging has completed
# ---------------------------------------------------------------------

proc cwreg_StopDragIconImage { startW x y startType } {
   set dragState [localDesktop get dragging]
   if { $dragState == "nodragging" } { return }

   set endType [cwreg_GetWidgetUnderMouse $startW $x $y $startType]
   set w [localDesktop get highlightWidget]

   $dragState highlight reset all
   . configure -cursor {}

   if { $w != "" } {
      $w setstate -bg grey
      cw_Debug "dropped icon $startW onto icon $w" cyan
      set startNode [cwreg_WidgetToStruct $startW]
      set endNode [cwreg_WidgetToStruct $w]

      if { $startType == $endType } {

         if { $startType == "team" } {
            cwreg_AddDetailsTo$endType                                  \
                              $endNode $startNode $startType team

         } else {
            #
            # we're allowed to make a subgroup of the icon
            cwreg_MakeNodeChild $endNode $startNode $startType
         }
      } else {
         #
         # add the dragged node's details to the dropped-on node

         cw_Debug "store details of $startNode in $endNode" cyan
         cwreg_AddDetailsTo$endType                                     \
                           $endNode $startNode $startType member

      }
   }
   localDesktop set dragging nodragging
   localDesktop set highlightWidget ""
}

# ---------------------------------------------------------------------
# Find out what widget is currently under the mouse, highlighting it if
```

```
# required. This routine uses the following rules:
#   valid highlighted nodes are:
#     orig icon = person, valid icons are teams and projects
#     orig icon = team, valid icons are (non-sub)teams and projects
#     orig icon = project, valid icons are (non-sub)projects
#     The original icon also cannot be highlighted
#     The original icon must also be the ROOT of its structure
#     (ie parent=null). If not, a break with the parent would occur
#     automatically - a feature we probably don't want ??? Anyway,
#     links can be explicitly broken instead to allow for the same
#     feature.
#
# If the icon is allowed, it will be stored in the localDesktop
# highlightWidget environment
# Returns the type of the drop is onto the same icon type,
# otherwise 0 (eg. team onto a team)
#
# NOTE: There's a lot of hard-coding in here for the groupIcon widget,
# making this code very non-portable, but is due to the Tk binding
# mechanism (where enter/leave events don't occur when a drag is
# occuring for another widget)
#   ??? maybe redo in some other way (some other time) ???
#   ??? Also uses stuff stored in the groupIcon envir -ie the icon type
# ------------------------------------------------------------------------

proc cwreg_GetWidgetUnderMouse { startW x y origType } {
   set w [winfo containing $x $y]
   if { $w == "" } { return }

   set wClass [winfo class $w]
   set droppedOnIcon "nothing"
   set currentHighlight [localDesktop get highlightWidget]
   set icnType ""

   #
   # first find out if we are on a groupIcon widget. First we check the
   # frame, and then the parent (if the widget was a label) to see if
   # it was a groupIcon. This is pretty horrible code - there's got to
   # be a better way!

   if { $wClass == "nodeIconClass" } { set droppedOnIcon $w
   } elseif { $wClass == "Label" } {

      #
      # totally major fudge - we check if the parent is a groupIcon and
      # if it is set our widget to be it

      set parent [winfo parent $w]
      if { [winfo class $parent] == "nodeIconClass" } {
         set droppedOnIcon $parent
      }
   }

   #
   # get rid of whatever widget is currently highlighted

   if {($currentHighlight !=$droppedOnIcon)&&($currentHighlight !="")}{
      $currentHighlight setstate -bg grey
   }
   localDesktop set highlightWidget ""

   #
   # time to highlight the widget we are over. This code uses the rules
   # mentioned above to determine whether the widget is valid or not

   if {($droppedOnIcon != "nothing") && ($droppedOnIcon != $startW)} {
      set env [$droppedOnIcon getenvironment]
```

```
        set icnType [$env get icontype]
        set startNode "$origType.[$startW cget -envname]"
        set endNode "$icnType.[$droppedOnIcon cget -envname]"

        set allowed 0
        if { $origType == "people" } {
           if { $icnType != "people" } { set allowed 1 }
        } elseif { ($origType == "team") && ($icnType == "proj") } {
           set allowed 1
        } elseif { ($origType == "team") && ($icnType == "team") } {

           #
           # previously teams were hierarchical like projects. Now they
           # are set-based like people. Just make sure the team we are
           # dropping onto does not contain the parent in one of its
           # child nodes

           set allowed 1

        } elseif { ($origType == "proj") && ($icnType == "proj") } {

           #
           # a project cannot become a subset of another team or
           # project unless it is the root project (ie it has no parent)
           # otherwise some automatic link break would need to take place

           if { [collaborwriter get $startNode.parent] == "" } {

              #
              # see if we're dropping on an icon in our hierarchy

              if { ![cwreg_SeeIfInHierarchy $startNode $endNode] } {
                 set allowed 1
              }
           }
        }

        if { $allowed == 1 } {
           $droppedOnIcon setstate -bg cyan
           localDesktop set highlightWidget $droppedOnIcon
           update idletasks
        }
     }
     return $icnType
}

# ---------------------------------------------------------------------
# Add a new icon (either team or proj or people) to the appropriate
# canvas. This routine is called as the result of an addEnvInfo event
# to the collaborwriter environment
# ---------------------------------------------------------------------

proc cwreg_AddIconToView {type iconName canvas icn} {
     set c [cwreg_GetGlobalCoords $type]                                \
     cwreg_CreateIcon
        $iconName $canvas $icn $type [lindex $c 0] [lindex $c 1]
}

# ---------------------------------------------------------------------
# Create a new icon - called from clicking OK on a CreateNewXXXX dialog
# ---------------------------------------------------------------------

proc cwreg_ClickedCreateOK { w type } {

     #
     # pull the entry name out of the dialog
```

```
    set entryName [$w.entry get]
    if [cwreg_MakeNewIcon $type $entryName] { cw_RemoveDialog $w }
}

# ------------------------------------------------------------------------
# Make a new icon
# Called whenever an event occurs where a NEW icon is to be created,
# eg. after a Create dialog or when a NEW person enters into our world
# ------------------------------------------------------------------------

proc cwreg_MakeNewIcon { type entryName } {

    #
    # See if the name already exists as an icon somewhere. This means we
    # can't use the same name for a project and a team, or for 2 people.
    # ??? In our startup, we'll need to check that people's names are
    # unique ???

    set original 1
    set widgetName [cw_RemoveSpacesFromName $entryName 1]
    foreach i [collaborwriter keys desktop.icons] {
        if { $i == $widgetName } { set original 0; break }
    }

    if { $original } {

        #
        # ??? MAJOR PROBLEM ???
        # I want to create a -transaction- of changes and pass them to
        # other RCs. I can't use events (they're only for conf processes)
        # and I can't use the keylist import facility, as that doesn't
        # update the server!! I have to do my creations carefully and
        # check for errors (eg trying to set a widget flag when its not
        # been created yet)

        # What's required is a CreateTransaction xx and SendTransaction xx
        # command which passes the info around everyone but only through
        # ONE event...  ??? DONE!!! - but not implemented here... ???

        #
        # this next line is pointless, as the environment won't store the
        # values initially anyway!
        # keylset data parent "" children ""

        #
        # get the icon and basic data structure created first

        set n "$type.$entryName"
        collaborwriter set $n.createFlag 1

        #
        # now add any extra info as required

        set icn "[set type]icon"

        switch $type {
            team    {

                # a new team, so make the originator the only member
                # (and a team leader at that)

                collaborwriter set $n.members
                        [list [list "people.[userprefs name]" leader]]

            }
            proj    {

                # a new project, so make the originator the creator
```

```
            collaborwriter set $n.members [list [list                    \
               "people.[userprefs name]" leader yes yes yes yes yes]]
            collaborwriter set $n.tools [list [list "Authoring tool"     \
               "/phd/gk/cw/tools/editor.tool"                            \
               "/phd/gk/cw/bitmaps/editor.gif"]]
         }
         people   {

            # a new person, so add their details to our environment
            # collaborwriter set $n.here 1

            if { $entryName == "Anybody" } { set icn "anonicon" }
         }
      }

      #
      # The above item will trigger the other RCs to create a new icon
      # at the standard x,y global position. Other changes to the
      # collaborwriter environment are made HERE instead of in the view
      # creation routines so that the changes won't be made (again) by
      # each RC instance

      #
      # and update the desktop icon list

      collaborwriter set desktop.icons.$widgetName                       \
         [list $icn ".$type.f.c" $entryName $type]

   } else {

      # the team/project name already exists, so just beep and keep
      # the dialog up

      cw_Debug "That name already exists!\a" yellow
      return 0
   }

   return 1
}

# ------------------------------------------------------------------------
# Create a simple dialog consisting of an entry +  OK and Cancel button
# ------------------------------------------------------------------------

proc cwreg_CreateEntryDialog { w } {

   if ![cw_CreateDialog $w] { return 0 }

   #
   # create the widgets for the dialog

   label $w.label
   entry $w.entry -relief sunken

   #
   # there are certain keypresses which ARN'T allowed in the dialog,
   # as they make bad widget elements or are reserved letters (eg "_")

   cwreg_BindWidget [list semicolon colon period comma underscore      \
      quotedbl backslash bracketleft bracketright braceleft braceright \
      percent] $w.entry

   #
   # pack the dialog onto the screen

   pack $w.label
   pack $w.entry -pady 10 -padx 30
```

```
    pack $w.cancel -side left -padx 10
    pack $w.ok -side right -padx 10

    return 1
}

# ----------------------------------------------------------------------
# Create a new team. At this stage the only thing to add is the name of
# the team. This process will create an icon for the team on the screen
# ----------------------------------------------------------------------

proc cwreg_CreateTeam {} {
    if ![cwreg_CreateEntryDialog .createteam] { return }
    wm title .createteam "Create a Team"

    #
    # modify the widgets for the dialog

    .createteam.label configure -text "The new team name is:"
    .createteam.ok configure
                -command "cwreg_ClickedCreateOK .createteam team"
}

# ----------------------------------------------------------------------
# Create a new project. Again the only thing added is the project name,
# which creates an icon on the screen.
# ----------------------------------------------------------------------

proc cwreg_CreateProject {} {
    if ![cwreg_CreateEntryDialog .createproj] { return }
    wm title .createproj "Create a Project"

    #
    # modify the widgets for the dialog

    .createproj.label configure -text "The new project name is:"
    .createproj.ok configure
                -command "cwreg_ClickedCreateOK .createproj proj"
}

# ----------------------------------------------------------------------
# Check if we can add a specified tool to the current project
# ----------------------------------------------------------------------

proc cwreg_ProjectToolToAdd { vars dlog w details} {
    upvar #0 $vars mvars

    set name [$w.f1.e get]
    set cmdline [$w.f2.e get]
    set icon [$w.f3.e get]

    if { $cmdline == "" } {
        cw_Debug "\aThe tool needs a command to run to work" yellow
        return
    }
    if { $name == "" } { set name [file tail $cmdline] }
    if { $icon == "" } { set icon toolicon }

    #
    # make sure the tool name is unique to the project

    if { $details == "" } {
        foreach i $mvars(tools) {
            if { [lindex $i 0] == $name } {
            cw_Debug "\aAlready a tool called $name in this project" yellow
            return
            }
```

```
      }
      lappend mvars(tools) [list $name $cmdline $icon]
   } else {
      set idx [lsearch $mvars(tools) $details]
      set mvars(tools) [lreplace $mvars(tools) $idx $idx                \
                        [list $name $cmdline $icon]]
   }

   #
   # get the tool details added to our system

   cwreg_UpdateModifyProjTools $vars $dlog
   cw_RemoveDialog $w
}

# ---------------------------------------------------------------------
# Dialog to determine if we can change a project tools details -
# runs off the back of the Add Tool dialog
# ---------------------------------------------------------------------

proc cwreg_ProjectToolChange { vars dlog details } {
   set w [cwreg_ProjectToolAdd $vars $dlog]

   if { $w != "" } {
      wm title $w "Change tool in project"
      cw_SetEntry $w.f3.e "[lindex $details 2]"
      cw_SetEntry $w.f2.e "[lindex $details 1]"
      cw_SetEntry $w.f1.e "[lindex $details 0]"
      $w.ok configure                                                   \
         -command [list cwreg_ProjectToolToAdd $vars $dlog $w $details]
   }
}

# ---------------------------------------------------------------------
# Dialog for whether we can add a tool to the current project
# ---------------------------------------------------------------------

proc cwreg_ProjectToolAdd { vars dlog } {
   set w "$dlog.add"
   if ![cw_CreateDialog $w] { return }

   wm title $w "Add tool to project"
   frame $w.f1; frame $w.f2; frame $w.f3
   label $w.f1.l -text "Tool:\t"
   label $w.f2.l -text "Location:\t"
   label $w.f3.l -text "Icon:\t"
   entry $w.f1.e -relief sunken
   entry $w.f2.e -relief sunken
   entry $w.f3.e -relief sunken
   button $w.f2.b -text "Browse" -command "cw_BrowseFiles $w.f2.e 0"
   button $w.f3.b -text "Browse" -command "cw_BrowseFiles $w.f3.e 1"
                                                                        \
   $w.ok configure
      -command [list cwreg_ProjectToolToAdd $vars $dlog $w ""]

   pack $w.f1 $w.f2 $w.f3 -side top -fill x -expand yes
   pack $w.cancel -side left -padx 10 -pady 10
   pack $w.ok -side right -padx 10 -pady 10

   pack $w.f1.l $w.f1.e -side left -anchor w
   pack $w.f2.l $w.f2.e $w.f2.b -side left -anchor w
   pack $w.f3.l $w.f3.e $w.f3.b -side left -anchor w

   return $w
}

# ---------------------------------------------------------------------
# Environment change while displaying the modify proj... dialog
# ---------------------------------------------------------------------
```

```
proc cwreg_ModifyProjEnvChanged { dlog labelName key } {
   set parts [split $key .]
   set lastPart [lindex $parts end]

   if {($lastPart == "members") && ([lindex $parts 1] == $labelName)}{

      #
      # our membership list has changed for this particular object

      cwreg_ReadCollaborwriterEnv $dlog "{$dlog}mvars" tools
      cwreg_UpdateModifyProjDialog "{$dlog}mvars" $dlog
   } elseif {($lastPart =="tools")&&([lindex $parts 1] ==$labelName)} {

      #
      # the tools available has changed

      cwreg_ReadCollaborwriterEnv $dlog "{$dlog}mvars" tools
      cwreg_UpdateModifyProjTools "{$dlog}mvars" $dlog
   }
}

# ---------------------------------------------------------------------
# Special case for when the yes/no/ask option changes for joining a
# conference, as we want to disable the initial rights if you arn't
# allowed into the conference
# ---------------------------------------------------------------------

proc cwreg_ChgBtnVal { vars col row val w fromMenu } {
   upvar #0 $vars mvars
   set oldVal $mvars(cbArray$col-$row)
   boxscroll_ChgBtnVal $vars $col $row $val $w $fromMenu
   set newVal $mvars(cbArray$col-$row)

   if { ($oldVal == "no") || ($newVal == "no") } {
      if { $newVal == "no" } { set state disabled
      } else { set state normal }

      for { set n 3 } { $n < 7 } { incr n } {
         $w.cb$n-$row configure -state $state
      }
   }
}

# ---------------------------------------------------------------------
# Update the modify proj... tools dialog after we've received an event
# saying that its state has changed
# ---------------------------------------------------------------------

proc cwreg_UpdateModifyProjTools { vars dlog } {
   upvar #0 $vars mvars

   #
   # when the thing has been updated, we just throw away the changes
   # we've been making
   # ??????????????????????????????????????????????????????????

   set currentTools [info commands $dlog.t.tools.*]
   foreach i $currentTools {
      cw_Debug " ???? ModifyProjTools: tool = $i" red

      # ??? code not implemented to remove a widget that isn't in the
      # ???  mvars(tools) array any more (as from an envir. update)

   }

   foreach i $mvars(tools) {
      set name [cw_RemoveSpacesFromName [lindex $i 0] 1]
```

```
        if { [info commands $dlog.t.tools.$name] == "" } {
            groupIcon $dlog.t.tools.$name -icontext [lindex $i 0]          \
                -widget [lindex $i 2] -defaulticon [lindex $i 2]
            pack $dlog.t.tools.$name -side left -padx 2 -pady 2
            $dlog.t.tools.$name bindall <Double-1>                         \
                [list cwreg_ProjectToolChange $vars $dlog $i]
        }
    }
}

# ---------------------------------------------------------------------------
# Update the main modify proj... dialog after we've received an event
# saying that its state has changed
# ---------------------------------------------------------------------------

proc cwreg_UpdateModifyProjDialog { vars dlog } {
    upvar #0 $vars mvars

    boxscroll_UpdateDialog $vars $dlog

    #
    # see if we are a team leader (and can make any changes to the
    # dialog info)

    set myState
        [cwreg_SeeIfMemberOf $mvars(node) [userprefs name] justThisNode]

    if { $myState != "leader" } {
        catch { pack forget $dlog.l.remove }
        catch { pack forget $dlog.t.remove }
        catch { pack forget $dlog.t.add }
        set state disabled
    } else {
        catch { pack $dlog.l.remove -side left -padx 10 -pady 10 }
        catch { pack $dlog.t.remove -side left -padx 10 -pady 10 }
        catch { pack $dlog.t.add -side right -padx 10 -pady 10 }
        set state normal
    }
    for { set i 0 } { $i < $mvars(listSize) } { incr i } {
        $dlog.f.p.f$i.cb1-$i configure -state $state
        $dlog.f.p.f$i.cb2-$i configure -state $state
        set val $mvars(cbArray2-$i)

        if { $val == "no" } { set cbState disabled
        } else { set cbState normal }

        for { set j 3 } { $j < $mvars(numAttrs) } { incr j } {
            $dlog.f.p.f$i.cb$j-$i configure -state $cbState
        }
    }
}

# ---------------------------------------------------------------------------
# Modify an existing project
# ---------------------------------------------------------------------------

proc cwreg_Modifyproj { w widgetName labelName } {
    set dlog ".mpj_[cw_ChangeDotInName $w]"
    if ![cw_CreateDialog $dlog] { return }

    global "{$dlog}mvars"
    set "{$dlog}mvars(node)" [cwreg_WidgetToStruct $w]
    set "{$dlog}mvars(listSize)" 0
    set "{$dlog}mvars(listIndex)" 0
    set "{$dlog}mvars(removeList)" {}
    set "{$dlog}mvars(tools)" {}
```

```
wm title $dlog "Modify project \"$labelName\""

frame $dlog.f -bd 4 -relief ridge
frame $dlog.f.n; frame $dlog.f.p; frame $dlog.f.j; frame $dlog.f.r
frame $dlog.f.s
label $dlog.f.n.l -pady 0 -bd 0 -text "\nProject member"
label $dlog.f.p.l -pady 0 -bd 0                                          \
   -text "\t          Enter\t   ------ Initial Access Rights ------"
label $dlog.f.p.l2 -pady 3 -bd 0                                         \
   -text "    Status  Editor\tRead\tComment\t Write \tDelegate"
label $dlog.f.s.l -pady 0 -bd 0 -text "\n"
listbox $dlog.f.n.namelist -width 20 -selectmode extended                \
   -yscrollcommand [list boxscroll_ListChanged                           \
                    "{$dlog}mvars" $dlog $dlog.f.s.scrV set]
scrollbar $dlog.f.s.scrV -orient vertical                                \
   -command [list boxscroll_ListChanged                                  \
             "{$dlog}mvars" $dlog $dlog.f.n.namelist yview]

frame $dlog.l
button $dlog.l.remove -text "Remove from project" -wraplength 80         \
   -state disabled                                                       \
   -command [list cwreg_ModifyProjectRemove "{$dlog}mvars" $dlog]
button $dlog.l.icon -text "Change icon" -wraplength 50 -command          \
   [list cwreg_ChangeIconDialog $w $widgetName $labelName x proj]

#
# build up the widgets for the project tools

frame $dlog.t -bd 4 -relief ridge
label $dlog.t.l -text "Project Tools"
frame $dlog.t.tools -bd 2 -relief groove
button $dlog.t.remove -text "Remove tool" -wraplength 80                 \
   -state disabled                                                       \
   -command [list cwreg_ProjectToolRemove "{$dlog}mvars" $dlog]
button $dlog.t.add -text "Add tool" -wraplength 50                       \
   -command [list cwreg_ProjectToolAdd "{$dlog}mvars" $dlog]

#
# build up the menu entries for the items in the popup menus

set "{$dlog}mvars(menutable.1.items)" [list leader member]
set "{$dlog}mvars(menutable.2.items)" [list yes no ask]
set "{$dlog}mvars(menutable.3.items)" [list yes no inherit]
set "{$dlog}mvars(menutable.4.items)" [list yes no inherit]
set "{$dlog}mvars(menutable.5.items)" [list yes no inherit]
set "{$dlog}mvars(menutable.6.items)" [list yes no inherit]

for { set i 1 } { $i < 7 } { incr i } {
   set "{$dlog}mvars(menutable.$i.cmd)" "boxscroll_ChgBtnVal"
}
set "{$dlog}mvars(menutable.2.cmd)" "cwreg_ChgBtnVal"

#
# modify the default widgets
                                                                         \
$dlog.ok configure
   -command [list cwreg_ModifyProjOK "{$dlog}mvars" $dlog] -text OK
                                                                         \
$dlog.cancel configure
   -command [list cwreg_ModifyTeamCancel "{$dlog}mvars" $dlog]

cwreg_ReadCollaborwriterEnv $dlog "{$dlog}mvars" tools

#
# get the dialog placed on the screen

pack $dlog.f $dlog.l $dlog.t -fill x
pack $dlog.f.s $dlog.f.p -side right -fill y
pack $dlog.f.n -fill both
```

```
    pack $dlog.f.n.l $dlog.f.p.l $dlog.f.s.l -side top -anchor w
    pack $dlog.f.p.l2 -side top -anchor w

    pack $dlog.t.l $dlog.t.tools -side top -fill x
    pack $dlog.t.tools -fill x
    pack $dlog.t.remove -side left -padx 10 -pady 10
    pack $dlog.t.add -side right -padx 10 -pady 10

    cwreg_UpdateModifyProjDialog "{$dlog}mvars" $dlog
    cwreg_UpdateModifyProjTools "{$dlog}mvars" $dlog

    pack $dlog.f.n.namelist -fill both
    pack $dlog.f.s.scrV -side right -fill y
    pack $dlog.l.icon -side right -padx 10 -pady 10
    pack $dlog.cancel -side left -padx 10 -pady 10
    pack $dlog.ok -side right -padx 10 -pady 10

    #
    # set up any necessary bindings for the dialog

    set "{$dlog}mvars(oldBind)"                                              \
       "cw_ListboxBtn $dlog.f.n.namelist $dlog.l.remove"
    bind Listbox <Button-1>                                                  \
       +[list cw_ListboxBtn $dlog.f.n.namelist $dlog.l.remove]

    set "{$dlog}mvars(CW1bind)" [collaborwriter bind addEnvInfo              \
       [list cwreg_ModifyProjEnvChanged $dlog $labelName %K]]
    set "{$dlog}mvars(CW2bind)" [collaborwriter bind changeEnvInfo           \
       [list cwreg_ModifyProjEnvChanged $dlog $labelName %K]]
}

# ---------------------------------------------------------------------------
# Routine to copy changes made to the project to the environment
# ---------------------------------------------------------------------------

proc cwreg_ModifyProjOK { vars dlog } {
   upvar #0 $vars mvars

   #
   # first update the fields detailing the member attributes

   set collaborwriterChanged 0
   set origList [collaborwriter get $mvars(node).members]

   for { set i 0 } { $i < $mvars(totalMembers) } { incr i } {
      set idx [cw_ListSearch $origList 0 $mvars(nameArray$i)]
      set origInfo [lindex $origList $idx]

      for { set j 1 } { $j < $mvars(numAttrs) } { incr j } {
         if { $mvars(stateArray$j-$i) != [lindex $origInfo $j] } {
            set origInfo                                                     \
               [lreplace $origInfo $j $j $mvars(stateArray$j-$i)]
            set origList [lreplace $origList $idx $idx $origInfo]
            set collaborwriterChanged 1
         }
      }
   }

   if { $collaborwriterChanged == 1 } {
      collaborwriter set $mvars(node).members $origList
   }

   #
   # now update the set of tools available in the project

   set origTools [collaborwriter get $mvars(node).tools]
   if { $mvars(tools) != $origTools } {
```

```
      collaborwriter set $mvars(node).tools $mvars(tools)
      foreach i $mvars(tools) {
         userprefs "prog.[lindex $i 0].cmd"                                    \
                                    "exec gkwish -f [lindex $i 1]"
      }
   }

   cwreg_ModifyTeamCancel $vars $dlog
}

# ------------------------------------------------------------------------
# Remove a member from the project
# ------------------------------------------------------------------------

proc cwreg_ModifyProjectRemove { vars dlog } {
   cw_Debug "cwreg_ModifyProjRemove NOT IMPLEMENTED YET" red
}

# ------------------------------------------------------------------------
# Routines for listbox/checkbtn manipulation in the modify team dialog
# ------------------------------------------------------------------------

proc cwreg_ToggleCheckBtn { w i vars } {
   upvar #0 $vars mvars
   $w configure -text $mvars(cbArray$i)
   set mvars(stateArray1-[expr $mvars(listIndex)+$i]) $mvars(cbArray$i)
}

proc cwreg_ListChanged { vars dlog cmd cmd2 args } {
   upvar #0 $vars mvars

   eval "$cmd $cmd2 $args"
   set topy [lindex [$dlog.f.n.namelist yview] 0]
   set topIndex [expr int($mvars(totalMembers) * $topy)]
   cwreg_MoveCheckButtons $dlog $topIndex $vars
}

proc cwreg_MoveCheckButtons { dlog index vars } {
   upvar #0 $vars mvars

   if { $index == $mvars(listIndex) } { return }
   set mvars(listIndex) $index

   for { set i 0 } { $i < $mvars(listSize) } { incr i } {
      set state $mvars(stateArray1-[expr $index + $i])
      set onvalue [$dlog.f.l.cb$i cget -onvalue]

      if { $state == "team" } {
         if { $onvalue != "team" } {
            $dlog.f.l.cb$i configure -onvalue team -offvalue team
            $dlog.f.l.cb$i invoke
         }
      } else {
         if { $onvalue == "team" } {
            $dlog.f.l.cb$i configure -onvalue leader -offvalue member
            if { $mvars(stateArray1-[expr $index + $i]) == "member" } {
               $dlog.f.l.cb$i invoke
            }
            $dlog.f.l.cb$i invoke

         }

         if {$mvars(stateArray1-[expr $index +$i]) !=$mvars(cbArray$i)}{
            $dlog.f.l.cb$i invoke
         }
      }
   }
}
```

```
# ------------------------------------------------------------------------
# Click on the Remove... button to remove members from the team. The
# members are not removed until OK is clicked.
# Team leaders are not removed (or changed) automatically. Instead the
# change is 'requested' by other team leaders through a voting system.
# ------------------------------------------------------------------------

proc cwreg_ModifyTeamRemove { vars dlog } {
   upvar #0 $vars mvars

   set removeItems [$dlog.f.n.namelist curselection]
   set numItems [llength $removeItems]
   if { $removeItems == "" } {
      #
      # nothing is selected, and the button should have been dimmed,
      # so disable it

      cw_ListboxBtn $dlog.f.n.namelist $dlog.l.remove
   } elseif { $numItems == $mvars(totalMembers) } {
      #
      # we're trying to remove everyone in the list. This either
      # deletes the list or isn't allowed - we'll disallow for now ???

      cw_Debug "You can't delete all the team members !! \a" yellow
   } else {
      #
      # remove the members from our list

      set removeItems [lsort -decreasing $removeItems]

      foreach i $removeItems {
         lappend mvars(removeList) $mvars(nameArray$i)
         $dlog.f.n.namelist delete $i
      }
      set topItem [lindex $removeItems 0]
      for {set i [expr $topItem+1]} {$i<$mvars(totalMembers)} {incr i}{
         set mvars(nameArray[expr $i - $numItems]) $mvars(nameArray$i)
         set mvars(typeArray[expr $i - $numItems]) $mvars(typeArray$i)
         for { set j 1 } { $j < $mvars(numAttrs) } { incr j } {
            set mvars(stateArray$j-[expr $i - $numItems])              \
                                          $mvars(stateArray$j-$i)
            unset mvars(stateArray$j-$i)
         }
         unset mvars(typeArray$i)
      }
      incr mvars(totalMembers) -$numItems

      $dlog.f.n.namelist selection clear 0 end
      cwreg_UpdateModifyTeamDialog $vars $dlog
      cw_ListboxBtn $dlog.f.n.namelist $dlog.l.remove
   }
}

# ------------------------------------------------------------------------
# Called as the result of an OK button click on the modify team dialog
# ------------------------------------------------------------------------

proc cwreg_ModifyTeamOK { vars dlog } {
   upvar #0 $vars mvars

   #
   # first make sure there's at least one team leader

   set ok 0
   for { set i 0 } { $i < $mvars(totalMembers) } { incr i } {
```

```
        if { $mvars(stateArray1-$i) == "leader" } { set ok 1 }
    }

    if { $ok == 0} {
        cw_Debug "Needs to be at least 1 leader in a team! \a" yellow
        return
    }

    #
    # Now we go and change our collaborwriter structure.

    set collaborwriterChanged 0
    set origList [collaborwriter get $mvars(node).members]
    set me "people.[userprefs name]"
    set tName [lindex [split $mvars(node) .] 1]

    # First remove any team members who are being deleted from the list,
    # or ask to remove any team leaders as required (unless it's
    # ourselves - we can kill ourselves OK!)

    foreach i $mvars(removeList) {
       set idx [cw_ListSearch $origList 0 $i]
       if { $idx != -1 } {

          #
          # the person still exists, so let's get them removed

          set origInfo [lindex $origList $idx]
          if {([lindex $origInfo 1]=="leader") &&                          \
              ([lindex $origInfo 0]!= $me)} {
             set p [lindex [split [lindex $origInfo 0] .] 1]
             cwreg_CreateVotingIssue
                [cwreg_GetLeaderList $mvars(node) justThisNode]           \
                "Request to remove leader $p from team: $tName"           \
                [list keep remove] [list {} [list cwreg_RemovePerson      \
                [lindex $origInfo 0] $mvars(node)]]                       \
                majority public public remove
          } else {

             #
             # remove the member from the collaborwriter environment

             set origList [lreplace $origList $idx $idx]
             set collaborwriterChanged 1

#         if { [lindex $origInfo 1] == "team" } {
#            #
#            # we got rid of a sub-item, so get the link broken to it
#            #
#            cwreg_RemoveChildParentLinkNode                               \
#                [lindex $origInfo 0] $mvars(node)
#         }

          }
       }
    }

    # Next we want to change the status of any team members who are now
    # leaders, or ask whether a team leader can be made into a member -
    # unless its ourself - we can make ourself a member if we want!

    for { set i 0 } { $i < $mvars(totalMembers) } { incr i } {
       set idx [cw_ListSearch $origList 0 $mvars(nameArray$i)]
       set origInfo [lindex $origList $idx]
       if { $mvars(stateArray1-$i) == "leader" } {
          if { [lindex $origInfo 1] != "leader" } {
             set origList [lreplace $origList $idx $idx                   \
```

```
                        [list [lindex $origInfo 0] leader]]
            set collaborwriterChanged 1
         }
      } else {
         if {([lindex $origInfo 1]=="leader") &&                    \
             ([lindex $origInfo 0]!= $me)} {

            #
            # we want to make a leader into a member!

            set p [lindex [split [lindex $origInfo 0] .] 1]
            cwreg_CreateVotingIssue
               [cwreg_GetLeaderList $mvars(node) justThisNode]      \
               "Request to change $p from a leader into a member of \
               team: $tName"                                        \
               [list leader member] [list {} [list                  \
               cwreg_DemotePerson [lindex $origInfo 0] $mvars(node)]] \
               majority public public member
         } else {
            if { [lindex $origInfo 1] != "team" } {
               set origList [lreplace $origList $idx $idx           \
                                 [list [lindex $origInfo 0] member]]
               set collaborwriterChanged 1
            }
         }
      }
   }

   if { $collaborwriterChanged == 1 } {
       collaborwriter set $mvars(node).members $origList
   }

   cwreg_ModifyTeamCancel $vars $dlog
}

# ---------------------------------------------------------------------
# Called as the result of a voting decision to remove a person. The
# person may have departed already, so check for this!
# ---------------------------------------------------------------------

proc cwreg_RemovePerson { name node } {
   set origList [collaborwriter get $node.members]
   set idx [cw_ListSearch $origList 0 $name]

   if { $idx != -1 } {

      #
      # the person still exists, so get them removed

      set origList [lreplace $origList $idx $idx]
      collaborwriter set $node.members $origList

#       if { [lindex $origInfo 1] == "team" } {
#          #
#          # we got rid of a sub-item, so get the link broken to it
#          cwreg_RemoveChildParentLinkNode \
#              [lindex $origInfo 0] $mvars(node)
#       }

   }
}

# ---------------------------------------------------------------------
# Called as the result of a voting decision to change a person from
# leader to member status.
# ---------------------------------------------------------------------
```

```
proc cwreg_DemotePerson { name node } {
    set origList [collaborwriter get $node.members]
    set idx [cw_ListSearch $origList 0 $name]
    if { $idx != -1 } {
        #
        # the person still exists, so get them demoted to member status
        set origList [lreplace $origList $idx $idx [list $name member]]
        collaborwriter set $node.members $origList
    }
}

# ------------------------------------------------------------------------
# Send an event to other collaborwriter people.
# Similar to gkEvent, but allows for users NOT being present and thus
# works ascynchronously and persistently
# ------------------------------------------------------------------------

proc cwreg_PostEvent { type data to } {
    set name [userprefs name]
    set eventData [list $type $data $name $to]
    #
    # before we post the event, see if we have already sent the same
    # event. If we have, just quit out of the routine and send nothing
    # more. This only checks for IDENTICAL events in the sense that no
    # people have received any of them yet. I.e. if a user sends an
    # event to 5 people and 1 deals with it, and the user sends the same
    # event to the same 5 people, then it will be regarded as a different
    # event and the four remaining users will receive 2 equivalent
    # events. ???

    # ??? Maybe the checks should be made at the receiving end, where a
    # user can check if there are 2 outstanding events (which are
    # equivalent) waiting for them- this doesn't help in the case where
    # the receiver is on-line, and will receive both events. ???

    set events [collaborwriter keys cwevent]
    foreach i $events {
        if [string match $name* $i] {
            set e [collaborwriter get cwevent.$i]
            if { $e == $eventData } { return }
        }
    }

    collaborwriter set cwevent.$name[cw_GetCounter] $eventData
}

# ------------------------------------------------------------------------
# Called as the result of cancelling the modify team dialog.
# This resets the widget bindings as appropriate for the collaborwriter
# environment and Listbox widget.
# ------------------------------------------------------------------------

proc cwreg_ModifyTeamCancel { vars dlog } {
    upvar #0 $vars mvars

    collaborwriter delbind $mvars(CW1bind)
    collaborwriter delbind $mvars(CW2bind)
    set binding [bind Listbox <Button-1>]
    regsub $mvars(oldBind) $binding "" binding
    bind Listbox <Button-1> $binding

    unset mvars
    cw_RemoveDialog $dlog
}
```

```
# ------------------------------------------------------------------------
# Update the modify team dialog currently displayed. This involves
# updating the listbox, member status, and checkbutton widgets.
# ------------------------------------------------------------------------

proc cwreg_UpdateModifyTeamDialog { vars dlog } {
   upvar #0 $vars mvars

   set oldSize $mvars(listSize)
   set mvars(listSize) [cw_Min $mvars(totalMembers) 5]

   if { $mvars(listSize) != $oldSize } {

      #
      # the list has changed size so we need to add/delete some
      # checkbuttons

      if { $mvars(listSize) > $oldSize } {
         for { set i $oldSize } { $i < $mvars(listSize) } { incr i } {
            set mvars(cbArray$i) leader
            checkbutton $dlog.f.l.cb$i -pady 0 -bd 0 -indicatoron 0     \
               -variable "{$dlog}mvars(cbArray$i)" -offvalue member     \
               -width 8 -onvalue leader -selectcolor #fae0fae0d2f1       \
               -bg #fae0fae0d2f1 -command [list cwreg_ToggleCheckBtn    \
               $dlog.f.l.cb$i $i "{$dlog}mvars"]
            if { $mvars(stateArray1-$i) == "member" } {
               $dlog.f.l.cb$i invoke
            } elseif { $mvars(stateArray1-$i) == "leader" } {
               $dlog.f.l.cb$i invoke; $dlog.f.l.cb$i invoke
            } else {
               $dlog.f.l.cb$i configure -offvalue team -onvalue team
               $dlog.f.l.cb$i invoke
            }
            pack $dlog.f.l.cb$i
         }
      } else {
         for {set i $oldSize} {$i > $mvars(listSize)} {incr i -1} {
            destroy $dlog.f.l.cb[expr $i - 1]
         }
      }

      #
      # now we need to set the listbox height and jump to the first
      # element

      $dlog.f.n.namelist configure -height $mvars(listSize)
      $dlog.f.n.namelist yview 0
      set idx 0
   } else {
      set idx $mvars(listIndex)
   }

   #
   # we just need to force the checkbuttons into being updated

   set mvars(listIndex) -1
   cwreg_MoveCheckButtons $dlog $idx $vars

   #
   # see if we are a team leader (and can make any changes to the dialog
   # info)

   set myState                                                          \
      [cwreg_SeeIfMemberOf $mvars(node) [userprefs name] justThisNode]

   if { $myState != "leader" } {
```

```
        catch { pack forget $dlog.l.remove }
        set state disabled
    } else {
        catch { pack $dlog.l.remove -side left -padx 10 -pady 10 }
        set state normal
    }

    for { set i 0 } { $i < $mvars(listSize) } { incr i } {
        $dlog.f.l.cb$i configure -state $state
    }
}

# ------------------------------------------------------------------------
# Environment change while displaying the modify... dialog
# ------------------------------------------------------------------------

proc cwreg_ModifyEnvChanged { dlog labelName key } {
    set parts [split $key .]
    set lastPart [lindex $parts end]

    if {($lastPart == "members") && ([lindex $parts 1] == $labelName)} {
        #
        # our membership list has changed for this particular object

        cwreg_ReadCollaborwriterEnv $dlog "{$dlog}mvars" ""
        cwreg_UpdateModifyTeamDialog "{$dlog}mvars" $dlog
    }
}

# ------------------------------------------------------------------------
# Read in the environment details for use in the node dialogs
# ------------------------------------------------------------------------

proc cwreg_ReadCollaborwriterEnv { dlog vars otherVars } {
    upvar #0 $vars mvars

    set node $mvars(node)
    set memberList [collaborwriter get $node.members]
    set numAttrs [llength [lindex $memberList 0]]

    foreach k $otherVars {
        set mvars($k) [collaborwriter get $node.$k]
    }

    set totalMembers 0
    $dlog.f.n.namelist delete 0 end

    foreach i $memberList {
        set item [lindex $i 0]
        if { [lsearch $mvars(removeList) $item] == -1 } {
            set names [split $item .]
            switch [lindex $names 0] {
                people   { set type "(person)" }
                team     { set type "(team)" }
                proj     { set type "(project)" }
            }
            $dlog.f.n.namelist insert end "[lindex $names 1] $type"
            set mvars(nameArray$totalMembers) $item
            for { set j 1 } { $j < $numAttrs } { incr j } {
                set mvars(stateArray$j-$totalMembers) [lindex $i $j]
            }
            set mvars(typeArray$totalMembers) [lindex $names 0]

            incr totalMembers
        }
    }
```

```
    set mvars(totalMembers) $totalMembers
    set mvars(memberList) $memberList
    set mvars(numAttrs) $numAttrs
}

# ----------------------------------------------------------------------
# Modify an existing team
# ----------------------------------------------------------------------

proc cwreg_Modifyteam { w widgetName labelName } {

    set dlog ".mtm_[cw_ChangeDotInName $w]"
    if ![cw_CreateDialog $dlog] { return }

    #
    # use a local array 'cbArray' to hold the checkbutton values
    # use a local array 'stateArray' to hold the state of all items in
    # the list and have a var linked to the top displayed item in list

    global "{$dlog}mvars"
    set "{$dlog}mvars(node)" [cwreg_WidgetToStruct $w]
    set "{$dlog}mvars(listSize)" 0
    set "{$dlog}mvars(listIndex)" 0
    set "{$dlog}mvars(removeList)" {}

    #
    # build up the dialog widgets, based on a listbox with checkbutton
    # icons to its side

    wm title $dlog "Modify team \"$labelName\""

    frame $dlog.f; frame $dlog.f.l; frame $dlog.f.n; frame $dlog.f.s
    label $dlog.f.l.l -text Status
    label $dlog.f.n.l -text "Team members"
    label $dlog.f.s.l
    listbox $dlog.f.n.namelist -width 35 -selectmode extended          \
       -yscrollcommand [list cwreg_ListChanged                         \
       "{$dlog}mvars" $dlog $dlog.f.s.scrV set]
    scrollbar $dlog.f.s.scrV -orient vertical -command [list           \
       cwreg_ListChanged "{$dlog}mvars" $dlog $dlog.f.n.namelist yview]

    menubutton $dlog.mbtn -text "Show all team members"                \
       -menu $dlog.mbtn.m -relief raised
    menu $dlog.mbtn.m -tearoff 0 -postcommand [list                    \
       cwreg_MakeMemberMenu $dlog.mbtn.m [set "{$dlog}mvars(node)"]]
    $dlog.mbtn.m add command -label "temp item"

    cwreg_ReadCollaborwriterEnv $dlog "{$dlog}mvars" ""

    frame $dlog.l
    button $dlog.l.remove -text "Remove from team" -wraplength 80      \
       -state disabled                                                 \
       -command [list cwreg_ModifyTeamRemove "{$dlog}mvars" $dlog]
    button $dlog.l.icon -text "Change icon" -wraplength 50 -command    \
       [list cwreg_ChangeIconDialog $w $widgetName $labelName x team]

    #
    # modify the default widgets

    $dlog.ok configure                                                 \
       -command [list cwreg_ModifyTeamOK "{$dlog}mvars" $dlog] -text OK
    $dlog.cancel configure                                             \
       -command [list cwreg_ModifyTeamCancel "{$dlog}mvars" $dlog]

    #
    # get the dialog placed on the screen

    pack $dlog.f $dlog.mbtn $dlog.l -fill x
    pack $dlog.f.s $dlog.f.l -side right -fill y
    pack $dlog.f.n -fill both
```

```
    pack $dlog.f.l.l $dlog.f.n.l $dlog.f.s.l -side top -fill x

    cwreg_UpdateModifyTeamDialog "{$dlog}mvars" $dlog

    pack $dlog.f.n.namelist -fill both
    pack $dlog.f.s.scrV -side right -fill y
    pack $dlog.l.icon -side right -padx 10 -pady 10
    pack $dlog.cancel -side left -padx 10
    pack $dlog.ok -side right -padx 10

    #
    # set up any necessary bindings for the dialog

    set "{$dlog}mvars(oldBind)"                                          \
       "cw_ListboxBtn $dlog.f.n.namelist $dlog.l.remove"
    bind Listbox <Button-1>                                              \
       +[list cw_ListboxBtn $dlog.f.n.namelist $dlog.l.remove]

    set "{$dlog}mvars(CW1bind)" [collaborwriter bind addEnvInfo          \
       [list cwreg_ModifyEnvChanged $dlog $labelName %K]]
    set "{$dlog}mvars(CW2bind)" [collaborwriter bind changeEnvInfo       \
       [list cwreg_ModifyEnvChanged $dlog $labelName %K]]
}

# ---------------------------------------------------------------------
# Modify an existing person. Anybody can change the person's icon
# locally on their own desktop, but can only apply the change globally
# if they are that person
# ---------------------------------------------------------------------

proc cwreg_Modifypeople { w widgetName labelName} {
   if { $labelName == [userprefs name] } { set state global
   } else { set state local }

   cwreg_ChangeIconDialog $w $widgetName $labelName $state people
}

# ---------------------------------------------------------------------
# Dialog for changing an icon on the desktop. This is really a
# Properties... dialog result
# ---------------------------------------------------------------------

proc cwreg_ChangeIconDialog {w widgetName labelName applyTo dlogType} {
   set dlog ".modifyicon_[cw_ChangeDotInName $w]"
   set defaultIcon                                                       \
      [lindex [collaborwriter get desktop.icons.$widgetName] 0]
   if ![cwreg_CreateEntryDialog $dlog] { return }
   set node [cwreg_WidgetToStruct $w]

   #
   # modify the widgets for the dialog (allow the "." character now for
   # file extns!)

   button $dlog.browse -text "Browse Icons" -wraplength 2c              \
      -justify center -command "cw_BrowseFiles $dlog.entry 1"
   button $dlog.globalOK -text "Change globally" -wraplength 2c         \
       -justify center -command "cwreg_ChangeIconOK $w $widgetName      \
       [list $labelName] $dlog global"
   button $dlog.pickDefault -text "Select global icon"                  \
      -command "cw_SetEntry $dlog.entry $defaultIcon"

   pack forget $dlog.ok

   bind $dlog.entry . {}
   wm title $dlog "Modify \"$labelName\""
   $dlog.label configure                                                \
```

```
      -text "Enter a new file/image name for the icon:"
   $dlog.ok configure -text "Change locally" -wraplength 2c              \
      -justify center -command "cwreg_ChangeIconOK $w $widgetName        \
      [list $labelName] $dlog local"
   $dlog.cancel configure -command [list cwreg_ChangeIconCancel $dlog]
   cw_SetEntry $dlog.entry [$w cget -widget]

   pack $dlog.browse -side right
   pack $dlog.pickDefault -side top -fill x
   pack $dlog.globalOK $dlog.ok -side right

   if { $dlogType != "people" } {
      global "{$dlog}mvars"
      set "{$dlog}mvars(CW1bind)" [collaborwriter bind addEnvInfo        \
         [list cwreg_IconEnvChanged $dlog $labelName $node %K]]
      set "{$dlog}mvars(CW2bind)" [collaborwriter bind changeEnvInfo     \
         [list cwreg_IconEnvChanged $dlog $labelName $node %K]]
      cwreg_SeeIfCanApplyIconGlobally $dlog $node
   } elseif { $applyTo != "global" } {
      $dlog.globalOK configure -state disabled
   }
}

# ---------------------------------------------------------------------
# Routine to allow for icon event changes in the icon dialog
# ---------------------------------------------------------------------

proc cwreg_IconEnvChanged { dlog labelName node key } {
   set parts [split $key .]
   set lastPart [lindex $parts end]

   if {($lastPart == "members") && ([lindex $parts 1] == $labelName)} {

      #
      # our membership list has changed for this particular object, so
      # see if we are still a member of the object.
      # For teams, any team member/leader can apply changes globally.
      # Icon creation from the people dialog will not call this routine

      cw_Debug "IconEnvChanged: $dlog $key" green
      cwreg_SeeIfCanApplyIconGlobally $dlog $node
   }
}

# ---------------------------------------------------------------------
# Routine to see if we are the owner of a particular icon
# ---------------------------------------------------------------------

proc cwreg_SeeIfCanApplyIconGlobally { dlog node } {

   #
   # if we are in the member list, then we can apply the icon change
   # globally

   set member [cwreg_SeeIfMemberOf $node [userprefs name] justThisNode]
   if { $member == "notFound" } {
      set state disabled } else { set state normal }
   $dlog.globalOK configure -state $state
}

# ---------------------------------------------------------------------
# IMPORTANT ROUTINE
# Search recursively (or not) through our node member lists to see if
# we are a member or not. Keeps searching for a 'leader' entity first
# Returns: notFound | leader | <whatever is set>
# ---------------------------------------------------------------------
```

```
proc cwreg_SeeIfMemberOf { baseNode name whichNode } {
   return [_cwreg_SeeIfMemberOf $baseNode $name $whichNode ""]
}

proc _cwreg_SeeIfMemberOf { baseNode name whichNode parsed } {

   #
   # first see if we've already parsed this node
   # (check for circular links)

   if { [lsearch $parsed $baseNode] != -1 } { return "notFound" }
   lappend parsed $baseNode

   set memberList [collaborwriter get $baseNode.members]
   set status notFound

   foreach i $memberList {
      set node [split [lindex $i 0] .]
      if { [lindex $node 1] == $name } {
         set status [lindex $i 1]
         if { $status == "leader" } { return $status }
      } elseif { ([lindex $node 0] != "people") &&                    \
         ($whichNode != "justThisNode") } {
         set val [_cwreg_SeeIfMemberOf                                \
               [lindex $i 0] $name $whichNode $parsed]
         if { $val == "leader" } { return leader }
         if { $val != "notFound" } { set status $val }
      }
   }

   return $status
}

# ---------------------------------------------------------------------
# Given a node, see if a person is a member somewhere here or *higher*
# up the hierarchy. i.e., we recurse up the tree to the root.
# ---------------------------------------------------------------------

proc cwreg_SeeIfMemberUpwardsOf { baseNode name } {
   set memberList [collaborwriter get $baseNode.members]
   set status notFound

   foreach i $memberList {
      set node [split [lindex $i 0] .]
      if { [lindex $node 1] == $name } {
         set status $i
         if { [lindex $status 1] == "leader" } { return $status }
      } else {
         set parent [collaborwriter get $baseNode.parent]
         if { $parent != "" } {
            set val [cwreg_SeeIfMemberUpwardsOf $parent $name]
            if { [lindex $val 1] == "leader" } { return $val }
            if { $val != "notFound" } { set status $val }
         }
      }
   }

   return $status
}

# ---------------------------------------------------------------------
# Routine to return a list of members in a hierarchy of nodes, being
# just this node, this node down, or the whole node tree
# ---------------------------------------------------------------------

proc cwreg_GetMembersOf { baseNode whichNodes recurse_up} {
   set members [_cwreg_GetMembersOf $baseNode "" $whichNodes ""]
```

```
    if { $recurse_up == "recurse" } {
       set parent [collaborwriter get $baseNode.parent]
       if { $parent != "" } {
          set newmem [cwreg_GetMembersOf $parent $whichNodes $recurse_up]
          set members [cw_UnionPeopleLists $members $newmem]
       }
    }
    return $members
}

proc _cwreg_GetMembersOf {
                     baseNode peopleList whichNodes alreadyParsed } {
    #
    # first see if we've already parsed this node
    # (check for circular links)

    if { [lsearch $alreadyParsed $baseNode] != -1 } {return $peopleList}

    lappend alreadyParsed $baseNode
    set memberList [collaborwriter get $baseNode.members]

    foreach i $memberList {
       set node [split [lindex $i 0] .]
       set name [lindex $node 1]

       #
       # check here for 'Anybody', who actually isn't a proper member

       if { $name == "Anybody" } { continue }

       set type [lindex $node 0]
       set state [lindex $i 1]
       if { $type == "people" } {
          set found ""
          foreach j $peopleList {
             if { $name == [lindex $j 0] } {
                set found $j
                set idx [lsearch $peopleList $j]
                if { $state == "leader" } { set newValue $state
                } else { set newValue [lindex $j 1] }

                set peopleList [lreplace $peopleList $idx $idx          \
                   [list $name $newValue]]
                break
             }
          }

          if { $found == "" } { lappend peopleList [list $name $state] }
       } else {
          if { $whichNodes != "justThisNode" } {
             set peopleList [_cwreg_GetMembersOf                        \
                [lindex $i 0] $peopleList $whichNodes $alreadyParsed]
          }
       }
    }

    return $peopleList
}

# ---------------------------------------------------------------------
# Return a list of any teams we are a member of
# ---------------------------------------------------------------------

proc cwreg_GetTeamsAmMemberOf { name } {
    set teams {}
    set allTeams [collaborwriter keys team]
```

```
    foreach team $allTeams {
       set state [cwreg_SeeIfMemberOf "team.$team" $name justThisNode]
       if { $state != "notFound" } { lappend teams "$team" }
    }

    return $teams
}

# ---------------------------------------------------------------------
# Return a list of any projects this team is a member of
# ---------------------------------------------------------------------

proc cwreg_GetProjectsTeamMemberOf { name } {
    set projects {}
    set allProjects [collaborwriter keys proj]

    foreach project $allProjects {
       set state [cwreg_SeeIfMemberOf "proj.$project" $name allNodes]
       if { $state != "notFound" } { lappend projects "proj.$project" }
    }

    return $projects
}

# ---------------------------------------------------------------------
# Return a list of all leaders in a particular node(s)
# ---------------------------------------------------------------------

proc cwreg_GetLeaderList { baseNode whichNode } {

#    if { $whichNode != "justThisNode" } {
#       set baseNode [cwreg_GetRootNode $baseNode]
#    }

    set leaderList {}
    set peopleList [cwreg_GetMembersOf $baseNode $whichNode recurse]
    foreach i $peopleList {
       if { [lindex $i 1] == "leader" } {
          lappend leaderList [lindex $i 0]
       }
    }

    return $leaderList
}

# ---------------------------------------------------------------------
# Make a menu showing all the members in a node
# ---------------------------------------------------------------------

proc cwreg_MakeMemberMenu { m baseNode } {

#    set baseNode [cwreg_GetRootNode $baseNode]

    set peopleList [cwreg_GetMembersOf $baseNode allNodes recurse]
    $m delete 0 end
    foreach i $peopleList {
       $m add command -label "[lindex $i 0] ([lindex $i 1])"
    }
}

# ---------------------------------------------------------------------
# Change the icon picture, either locally to the desktop or globally
# to all
# ---------------------------------------------------------------------

proc cwreg_ChangeIconOK { w widgetName labelName dialog scope } {
    set fname [cw_ExpandFileName [$dialog.entry get]]
```

```
    if [cwreg_ChangeIconPicture $fname] {
       $w configure -widget $fname

       if { $scope == "global" } {

          #
          # if we want to apply the change globally, we need to update
          # our desktop icon details in the collaborwriter environment

          set info [collaborwriter get desktop.icons.$widgetName]
          set info [lreplace $info 0 0 $fname]
          collaborwriter set desktop.icons.$widgetName $info
       }

       cwreg_ChangeIconCancel $dialog
    } elseif {![file exists $fname]} {
       cw_Debug "File $fname does not exist" yellow
    }
}

# ---------------------------------------------------------------------
# Tidy up bindings or cancel of the change icon dialog
# ---------------------------------------------------------------------

proc cwreg_ChangeIconCancel { dlog } {
   global "{$dlog}mvars"
   catch { collaborwriter delbind [set "{$dlog}mvars(CW1bind)"] }
   catch { collaborwriter delbind [set "{$dlog}mvars(CW2bind)"] }
   catch { unset "{$dlog}mvars" }
   cw_RemoveDialog $dlog
}

# ---------------------------------------------------------------------
# Change the icon picture to the one given in 'fname'
# ---------------------------------------------------------------------

proc cwreg_ChangeIconPicture { fname } {
   if { [cw_CreateImage [image types] $fname] == 1 } {
      return 1
   }

   cw_Debug "File $fname is not a valid picture file \a" yellow
   return 0
}

# ---------------------------------------------------------------------
# Given a person's name, find their respective icon definition from
# the desktop
# ---------------------------------------------------------------------

proc cwreg_FindIconPictureFor { name } {
   set widgetName [cw_RemoveSpacesFromName $name 1]
   set icn [collaborwriter get desktop.icons.$widgetName]
   return [lindex $icn 0]
}

# ---------------------------------------------------------------------
# Set up the user interface view. This consists of a row of buttons
# along the top, and three scrollable canvases for the projects, teams,
# and people in our world.
# ---------------------------------------------------------------------

proc cwreg_CreateView {} {

   #
   # initialise the window, loading in the default image details for
   # proj/team/people
```

```
wm minsize . 100 100
wm title . "CollaborWriter Project Administrator"
set i [image types] ; # ??? we could set this to bitmap ???
if {![cw_CreateNamedImage $i bitmaps/teamicon teamicon]} {
   error "No teamicon icon"}
if {![cw_CreateNamedImage $i bitmaps/projicon projicon]} {
   error "No projicon icon"}
if {![cw_CreateNamedImage $i bitmaps/peopleicon peopleicon]} {
   error "No peopleicon icon"}
if {![cw_CreateNamedImage $i bitmaps/anonicon anonicon]} {
   error "No anonicon icon"}
if {![cw_CreateNamedImage $i bitmaps/toolicon toolicon]} {
   error "No toolicon icon"}

label .topName -text "[userprefs name]"                                    \
               -relief raised -bg blue -fg white

#
# create a row of buttons to represent the diff. actions available

frame .bbar -relief groove -borderwidth 4
button .bbar.createTeam -text "Create new team" -wraplength 2c        \
   -justify center -padx 1c -relief raised                             \
   -command "cwreg_CreateTeam" -width 10
button .bbar.createProj -text "Create new project" -wraplength 3c     \
   -justify center -padx 1c -relief raised                             \
   -command "cwreg_CreateProject" -width 10
button .bbar.votingTool -text "Voting tool" -wraplength 3c            \
   -justify center -padx 1c -relief raised                             \
   -command "cwreg_OpenVotingTool" -width 10
button .bbar.quit -text "Exit without saving" -wraplength 2.5c        \
   -justify center -padx 10 -relief raised                             \
   -command "cwreg_Exit quit" -width 10
button .bbar.exit -text "Save desk and exit"  -wraplength 2.5c        \
   -justify center -padx 10 -relief raised                             \
   -command "cwreg_Exit save" -width 10

#
# create the three canvases used:
# .proj = projects, .team = teams, .people = people

foreach i { proj team people } {
   frame .$i -relief groove -borderwidth 2
   label .$i.label -relief raised -justify center                      \
                   -bg indianred -fg yellow
   frame .$i.f
   canvas .$i.f.c -width 4c -height 7c -scrollregion "0 0 1000 1000"\
      -yscrollcommand ".$i.scrV set" -xscrollcommand ".$i.scrH set"
   scrollbar .$i.scrV -orient vertical -command ".$i.f.c yview"
   scrollbar .$i.scrH -orient horizontal -command ".$i.f.c xview"
}

#
# plus the buttons, we can double-click to create a project/team

bind .proj.f.c <Double-1> "cwreg_CreateProject"
bind .team.f.c <Double-1> "cwreg_CreateTeam"

.proj.f.c configure -width 10c
.proj.label configure -text "Projects"
.team.label configure -text "Teams"
.people.label configure -text "People"

#
# get all the widgets drawn on the screen
```

```
    pack .topName -side top -fill x
    pack .bbar -side top -fill x
    pack .bbar.createProj .bbar.createTeam .bbar.votingTool -side left
    pack .bbar.exit .bbar.quit -side right
    pack .proj .team .people -side left -fill both -expand yes

    foreach i { proj team people } {
       pack .$i.label -side top -fill x
       pack .$i.scrV -side right -fill y
       pack .$i.scrH -side bottom -fill x
       pack .$i.f -side top -fill both -expand yes
       pack .$i.f.c -side top -fill both -expand yes
    }
}

# ------------------------------------------------------------------------
# Create a new anonymous user (called Anybody)
# ------------------------------------------------------------------------

proc cwreg_CreateAnonymousUser {} {
   cwreg_MakeNewIcon people Anybody
}

# ------------------------------------------------------------------------
# Create an issue for voting on. This remains persistent, even after the
# vote has been cast.
# Voting records are stored in collaborwriter vote.issuekey <data> where
# <data> is:
#  voters     A list of voters, with their decision - {} if undecided.
#  subject    The issue to vote on
#  comments   A list of comments by members able to vote on the subject
#  choices    A list of possible choices available to a voter
#  commands   A list of possible commands to execute depending on the
#             chosen vote
#  method     How a result should be decided
#             (unanimous / majority / individual)
#  people     How the people appear (anonymous / public)
#  ballot     How the ballot is performed (secret / public)
#  event      The event which has caused a change to the dialog
# ------------------------------------------------------------------------

proc cwreg_CreateVotingIssue {
     voterNames subject choices cmds method people ballot choice } {

   set issueNum [cwreg_GetVotingCounter]
   set voters {}

   lappend choices undecided
   lappend cmds {}
   foreach i $voterNames { lappend voters [list $i undecided] }

   #
   # change the data structure to include our preferred choice

   set idx [cw_ListSearch $voters 0 [userprefs name]]
   set voters                                                          \
      [lreplace $voters $idx $idx [list [userprefs name] $choice]]

   set data [list $voters $subject {} $choices $cmds $method $people   \
                  $ballot newIssue]

   collaborwriter set voting.$issueNum $data
   cw_Debug "voting issue $issueNum: $data" blue
   return $issueNum
}
```

```
# ------------------------------------------------------------------------
# Build up the display of all the current voting issues
# ------------------------------------------------------------------------

proc cwreg_GetVotingIssues { w } {
   global gVoteTool

   set issues [collaborwriter keys voting]

   foreach item $issues {
      set data [collaborwriter get voting.$item]
      set myIdx [cw_ListSearch [lindex $data 0] 0 [userprefs name]]
      if { $myIdx != -1 } {
         catch { destroy $w.f.b.b$item }
         radiobutton $w.f.b.b$item -indicatoron 1 -text "" -width 3     \
            -value $item  -variable gVoteTool(item)                     \
            -command "cwreg_FillInVoteFieldsFor $item $w"
         pack $w.f.b.b$item -side top
      } else {

         #
         # we arn't involved in this issue, so remove it from our list

         set idx [lsearch $issues $item]
         set issues [lreplace $issues $idx $idx]
      }
   }

   set numIssues [llength $issues]

   if { $gVoteTool(currentIssue) == -1 } {
      if { $numIssues > 0 } { $w.f.b.b[lindex $issues 0] invoke }
   }

   return $issues
}

# ------------------------------------------------------------------------
# Build up the dialog for our voting tool
# The voting routines should be in their own file ????
# ------------------------------------------------------------------------

proc cwreg_OpenVotingTool {} {
   global gVoteTool

   #
   # build up the user interface widgets for the voting dialog

   set w ".votingTool"
   if ![cw_CreateDialog $w] { return }
   wm title $w "Voting tool"
   set gVoteTool(currentIssue) -1

   frame $w.f; frame $w.f.b; frame $w.f.f; frame $w.f.f.subject
   frame $w.f.f.comments; frame $w.f.f.comments.t
   frame $w.f.f.comments.s
   frame $w.f.f.entry; frame $w.f.f.decisions -relief groove -bd 4

   label $w.f.f.subject.t -text "No issue to vote on" -justify center  \
      -bd 4 -relief groove -wraplength 10c
   scrollbar $w.f.f.comments.s.scrV -orient vertical
   text $w.f.f.comments.t.l -height 7 -wrap word -state disabled       \
      -yscrollcommand [list $w.f.f.comments.s.scrV set] -width 50     \
      -font *-lucida-medium-r-*-*-10-*-*-*-*-*-*-* -bg #fffffae0cdd2
   $w.f.f.comments.s.scrV configure -command "$w.f.f.comments.t.l yview"
   label $w.f.f.entry.l -text "Comment:"
   entry $w.f.f.entry.e -bg white -fg [userprefs color]
```

```
    button $w.f.f.entry.b -text "Add" -relief raised -bd 2              \
       -command "cwreg_AddVoteComment $w"
    label $w.f.f.decisions.l -text "Decisions"                          \
                             -justify center -relief raised -bd 4
    $w.cancel configure -text "Close"                                   \
       -command [list cwreg_CloseVotingTool $w]

    #
    # place all our widgets onto the screen

    pack $w.f -fill x -expand yes
    pack $w.cancel -side right -padx 10
    pack $w.f.b $w.f.f -side left -fill both -expand yes
    pack $w.f.f.subject $w.f.f.comments $w.f.f.entry $w.f.f.decisions    \
          -side top -fill both -expand yes
    pack $w.f.f.subject.t -fill x -expand yes
    pack $w.f.f.comments.t -side left -fill both
    pack $w.f.f.comments.s -side right -fill y
    pack $w.f.f.comments.t.l -fill x -expand yes
    pack $w.f.f.comments.s.scrV -fill both -expand yes
    pack $w.f.f.entry.l -side left
    pack $w.f.f.entry.b -side right
    pack $w.f.f.entry.e -fill x
    pack $w.f.f.decisions.l -fill x

    #
    # read out the number of issues, finding only those issues we are
    # involved in.

    set issues [cwreg_GetVotingIssues $w]

    # set numIssues [llength $issues]
    # if { $numIssues > 0 } { $w.f.b.b[lindex $issues 0] invoke }

    #
    # bind the collaborwriter environment so that we receive voting
    # changes. Do we look at other changes here? (e.g. a change in team
    # leaders, or the nullification of the thing to be voted on - eg. a
    # person removes themself when a vote is on to remove them  ???
    # We'll not handle it here, and just concentrate on passing comments
    # round the system

    set gVoteTool(CW1bind) [collaborwriter bind addEnvInfo              \
       [list cwreg_VoteEnvChanged $w %K]]
    set gVoteTool(CW2bind) [collaborwriter bind changeEnvInfo           \
       [list cwreg_VoteEnvChanged $w %K]]
}

# ---------------------------------------------------------------------
# Remove binding set up for the voting tool
# ---------------------------------------------------------------------

proc cwreg_CloseVotingTool { w } {
    global gVoteTool
    catch { collaborwriter delbind $gVoteTool(CW1bind) }
    catch { collaborwriter delbind $gVoteTool(CW2bind) }
    cw_RemoveDialog $w
}

# ---------------------------------------------------------------------
# Modify our vote environment as the world changes
# ---------------------------------------------------------------------

proc cwreg_VoteEnvChanged { w key } {
    global gVoteTool
    set parts [split $key .]
    set firstPart [lindex $parts 0]
```

```
    if { $firstPart == "voting"} {
        #
        # somethings changed with the voting stuff, so let's see what...
        set item [lindex $parts 1]
        set data [collaborwriter get $key]
        set event [lindex $data 8]
        if { $item == $gVoteTool(item) } {
            #
            # somethings changed on the voting thing we're looking at
            if { $event == "addComment" } {
                cwreg_InsertVoteComment
                    $w [lindex [lindex $data 2] end] [lindex $data 6]
            } elseif { $event == "decisionChange" } {
                cw_Debug "change by: [lindex $data 9]" blue
                cwreg_ShowDecisionOf
                    [lindex $data 9] [lindex $data 10] $w $data
            } elseif { $event == "issueClosed" } {
                set gVoteTool(currentIssue) -1
                cwreg_FillInVoteFieldsFor $item $w
            }
        } else {
            cwreg_GetVotingIssues $w
        }
    }
}

# ---------------------------------------------------------------------
# Insert a comment into the voting tool
# ---------------------------------------------------------------------

proc cwreg_InsertVoteComment { w comment person } {
    if { $person == "anonymous" } {
        set name ""
        set colour black
    } else {
        set name "[lindex $comment 0]: "
        set colour [lindex $comment 1]
    }
    set txt "$name[lindex $comment 2]"
    $w.f.f.comments.t.l tag configure $colour -foreground $colour
    $w.f.f.comments.t.l configure -state normal
    $w.f.f.comments.t.l insert end $txt [list $colour]
    $w.f.f.comments.t.l see end
    $w.f.f.comments.t.l configure -state disabled
}

# ---------------------------------------------------------------------
# Fill in all the voting details for when we change the voting item we
# are looking at
# ---------------------------------------------------------------------

proc cwreg_FillInVoteFieldsFor { item w } {
    global gVoteTool

    if { $gVoteTool(currentIssue) == $item } { return }
    set gVoteTool(currentIssue) $item
    set data [collaborwriter get voting.$item]
    if { [lindex $data 8] == "issueClosed" } {
        set state disabled} else {set state normal}

    $w.f.f.entry.e configure -state $state
    $w.f.f.entry.b configure -state $state
```

```
$w.f.f.subject.t configure -text [lindex $data 1]
set me [userprefs name]
set meWidget [cw_RemoveSpacesFromName $me 0]

#
# first get rid of all the old buttons in the decisions... list, and
# the text from the textbox display

set oldItems [split [info commands $w.f.f.decisions.f*] " "]
foreach i $oldItems { catch { destroy $i } }
$w.f.f.comments.t.l configure -state normal
$w.f.f.comments.t.l delete 1.0 end
$w.f.f.comments.t.l configure -state disabled

#
# get all our details for this issue into convenience variables

set voters [lindex $data 0]
set myIdx [cw_ListSearch $voters 0 $me]
set myDetails [lindex $voters $myIdx]
set voters [lreplace $voters $myIdx $myIdx]
set comments [lindex $data 2]
set options [lindex $data 3]
set people [lindex $data 6]
if { $people == "anonymous" } {
   set anonymous 1 } else { set anonymous 0 }
if { [lindex $data 7] == "public" } {
   set visiblevote 1 } else { set visiblevote 0 }

if { ($anonymous == 1) && ($visiblevote == 0) } {
   set totalAnon 1 } else { set totalAnon 0 }

#
# build up the list of decision-makers and their choices so far

set decisionMade 0

foreach i $voters {
   set name [cw_RemoveSpacesFromName [lindex $i 0] 0]
   if { $totalAnon == 0 } {
      frame $w.f.f.decisions.f$name
      label $w.f.f.decisions.f$name.l
      pack $w.f.f.decisions.f$name -fill x
      pack $w.f.f.decisions.f$name.l -side left -fill x -expand yes
   }
}

foreach i $voters {
   cwreg_ShowDecisionOf [lindex $i 0] [lindex $i 1] $w $data
}

#
# build up the options we have as a choice

frame $w.f.f.decisions.f$meWidget
label $w.f.f.decisions.f$meWidget.l -text "Your choice is:"
pack $w.f.f.decisions.f$meWidget -fill x -expand yes
pack $w.f.f.decisions.f$meWidget.l -side left -fill x -expand yes

foreach i $options {
   radiobutton $w.f.f.decisions.f$meWidget.b$i -text $i                \
      -indicatoron 0 -value $i                                         \
      -variable gVoteTool(choice$item) -selectcolor #ececec            \
      -command [list cwreg_VoteChoiceMade $item] -state $state
   pack $w.f.f.decisions.f$meWidget.b$i -side right
}
```

```
    set myChoice [lindex $myDetails 1]

    #
    # invoke our choice of decision - if only one user, this will cause
    # a decision to be made (being the majority/unanimous) and means
    # there isn't a chance to add any comments - also, only after opening
    # the voting tool will the decision be made, which seems a bit of a
    # fudge... ???

    $w.f.f.decisions.f$meWidget.b$myChoice invoke

    #
    # fill in the textbox with the details of the current comments

    foreach i $comments {
       cwreg_InsertVoteComment $w $i $people
    }
}

# ----------------------------------------------------------------------
# Chnage the button display to show the decision of user xxx
# ----------------------------------------------------------------------

proc cwreg_ShowDecisionOf { name decision w data } {
    if { $name == [userprefs name] } { return }

    set people [lindex $data 6]
    set ballot [lindex $data 7]

    #
    # first work out the final bit of the text

    if { $decision != "undecided" } {
       set decisionMade 1
       if { $ballot == "public" } {
          set endBit "chosen option: $decision"
       } else {
          set endBit "chosen an option"
       }
    } else {
       set decisionMade 0
       set endBit "yet to decide"
    }

    #
    # now work out who the text refers to

    if { $people == "anonymous" } {
       set startBit "One person has"
    } else {
       set startBit "$name has"
    }

    if { ($people == "anonymous") && ($ballot == "secret") } {
       return $decisionMade
    }

    #
    # finally update the widget associated with this item

    $w.f.f.decisions.f[cw_RemoveSpacesFromName $name 0].l                \
       configure -text "$startBit $endBit"
    return -1
}

# ----------------------------------------------------------------------
# Make a decision, so tell all & see if the vote issue has been closed
# ----------------------------------------------------------------------
```

```
proc cwreg_VoteChoiceMade { issue args } {
   global gVoteTool

   if { $args == "" } {
      set choice $gVoteTool(choice$issue) } else { set choice $args }

   cw_Debug "chosen: $choice for issue $issue" blue

   set me [userprefs name]
   set data [collaborwriter get voting.$issue]
   set voters [lindex $data 0]
   set myIdx [cw_ListSearch $voters 0 $me]
   set myDetails [lindex $voters $myIdx]

   if { [lindex $myDetails 1] != $choice } {

      #
      # we've made/changed a decision on an issue, so tell everyone else

      set myDetails [lreplace $myDetails 1 1 $choice]
      set voters [lreplace $voters $myIdx $myIdx $myDetails]
      set data [lrange $data 0 7]
      set data [lreplace $data 0 0 $voters]
      lappend data decisionChange $me $choice
      collaborwriter set voting.$issue $data
   }

   if { $choice == "undecided" } { return }

   #
   # finally, see if the choice made causes the vote discussion to
   # finish. Depending on the voting option, this will occur at
   # different times. i.e.
   # unanimous  - all people must decide on 1 outcome
   # majority   - most people must decide on 1 outcome
   # individual - any user can decide the outcome

   set method [lindex $data 5]
   set voteCast -1

   if { $method == "individual" } {
      set txt "??? issue decided for individual case"
      set voteCast 1
   } else {
      foreach i [lindex $data 3] { set numEntries($i) 0 }
      foreach person $voters { incr numEntries([lindex $person 1]) }

      set numPeople [llength $voters]
      set majority [expr $numPeople / 2]
      foreach i [lindex $data 3] {
         if { ($numEntries($i) > $majority) && ($i != "undecided") } {
            if { $method == "majority" } {
               set txt "majority decision $i picked!"
               set choice $i
               set voteCast 1
            } elseif { $numEntries($i) == $numPeople } {
               set txt "unanimous decision $i picked!"
               set choice $i
               set voteCast 1
            }
         }
      }
   }

   #
   # if the vote is complete, add a comment saying the issue is closed
   # and close the issue between all the voting participants
```

```
    if { $voteCast != -1 } {
        cw_Debug "issue closed" blue
        set comments [lindex $data 2]
        set txt "*** ISSUE CLOSED - $txt ***"
        lappend comments [list "" black $txt]
        set data [lreplace $data 2 2 $comments]
        set data [lreplace $data 8 8 addComment]
        collaborwriter set voting.$issue $data
        set data [lreplace $data 8 8 issueClosed]
        collaborwriter set voting.$issue $data

        #
        # execute the command which is linked with the decision

        set item [lsearch [lindex $data 3] $choice]
        set cmd [lindex [lindex $data 4] $item]
        if { $cmd != "" } {
            eval $cmd
        }
    }
}

# ----------------------------------------------------------------------
# Dialog for adding a comment to a voting issue
# ----------------------------------------------------------------------

proc cwreg_AddVoteComment { w } {
    global gVoteTool
    if { $gVoteTool(currentIssue) == -1 } {
        cw_Debug "There's no issue to add a comment to! \a" yellow
        return
    }

    set data [collaborwriter get voting.$gVoteTool(currentIssue)]

    set comments [lindex $data 2]
    set txt "[$w.f.f.entry.e get]\n"
    lappend comments [list [userprefs name] [userprefs color] $txt]
    set data [lreplace $data 2 2 $comments]
    set data [lreplace $data 8 8 addComment]
    collaborwriter set voting.$gVoteTool(currentIssue) $data
    $w.f.f.entry.e delete 0 end
}

# ----------------------------------------------------------------------
# Return a unique voting number
# ----------------------------------------------------------------------

proc cwreg_GetVotingCounter {} {
    return [cw_GetUniqueNum]
}

# ----------------------------------------------------------------------
# Exit the Collaborwriter registration client, saving our desktop in
# the process
# ----------------------------------------------------------------------

proc cwreg_Exit { saveDesktop } {
    global gDesktopFile

    keylset event type regClientQuitting
    gk_postEvent $event

    #
    # save our desktop settings to the file .cwdesktop

    if { $saveDesktop == "save" } {
        cw_Debug "Saving desktop settings..." cyan
```

```
        set fileID [open $gDesktopFile w]
        foreach type { proj team people } {
            puts $fileID [localDesktop get desktop.nextIconPos.$type]
        }

        set widgetList [collaborwriter keys desktop.icons]
        foreach i $widgetList {
            set w [collaborwriter get desktop.icons.$i]
            set c [lindex $w 1]
            set l [cw_RemoveSpacesFromName [lindex $w 2] 0]
            set e [$c.$i getenvironment]
            puts $fileID "$i [$e coords] [$c.$i cget -widget] $c $l [      \
                lindex $w 3] [lindex $w 0]"
        }
        close $fileID
    }

    _gk_properQuit
}

# ---------------------------------------------------------------------
# Initialisation routine, called once we have received the
# collaborwriter environment details from the central server
# ---------------------------------------------------------------------

proc cwreg_InitialiseSettings {} {
    global gDesktopFile
     cw_Debug "Got the collaborwriter env details from the server" cyan

    #
    # first read in our .cwdesktop resource file to see where WE have
    # positioned any icons

    if { [catch {set fileID [open $gDesktopFile r] }] == 0 } {
        foreach type { proj team people } {
            gets $fileID dataLine
            scan $dataLine "%f %f %f" x y column
            localDesktop set desktop.nextIconPos.$type [list $x $y $column]
        }

        gets $fileID dataLine
        while {![eof $fileID]} {
            scan $dataLine "%s %f %f %s %s %s %s %s"                    \
                widget x y icon c label type lastGlobalIcon
            set widgetName [cwreg_CreateIcon                            \
                [cw_AddSpacesToName $label] $c $icon $type $x $y]

            #
            # it's possible that global changes may have occurred while
            # we've been away, so see if they should be applied

            set winfo [collaborwriter get desktop.icons.$widgetName]
            if { $lastGlobalIcon != [lindex $winfo 0] } {

                # the global desktop icon is different, so see if we want to
                # change our current desktop icon

                cwreg_GlobalDesktopChange $widgetName $winfo
            }

            gets $fileID dataLine
        }
        close $fileID
    }

    #
    # other icons may have been created since we were last logged in, so
    # get those icons created as well
```

```
set widgetList [collaborwriter keys desktop.icons]
set newWidgets {}

foreach i $widgetList {
   set widgetInfo [collaborwriter get desktop.icons.$i]
   set w "[lindex $widgetInfo 1].$i"

   #
   # create the icons which we are missing (the collaborwriter
   # details are already set up, and just use the next localDesktop
   # coords position)

   if {[info commands $w]==""} {
      set type [lindex $widgetInfo 3]
      cwreg_NextGlobalCoords $type
      set xy [cwreg_GetGlobalCoords $type]
      cw_Debug "$w doesn't exist - creating with: $widgetInfo" cyan
      cwreg_CreateIcon [lindex $widgetInfo 2] [lindex $widgetInfo 1]\
         [lindex $widgetInfo 0] $type [lindex $xy 0] [lindex $xy 1]
      lappend newWidgets $w
   }
}

#
# look through the people to see if we are already created as a user

set myName [userprefs name]
set peopleList [collaborwriter keys people]
set alreadyPresent 0
foreach i $peopleList {
   if { $i == $myName } { set alreadyPresent 1; break }
}

if {! $alreadyPresent} {
   cw_Debug "need to add person $myName" cyan
   cwreg_AddUserColour $myName [userprefs color]
   cwreg_MakeNewIcon people $myName
}

#
# Check to see if the 'anybody' user exists. If not, create them.

if { [info commands ".people.f.c.anybody" ] == "" } {
   cwreg_CreateAnonymousUser
}

#
# Find any links which need to be made between the icons. To do this,
# first scan our icons for any with the parent="" field set (as
# they'll be root icons)

set rootIcons {}
foreach i { proj } {
   set icns [collaborwriter keys $i]
   foreach j $icns {
      if { [collaborwriter get $i.$j.parent] == "" } {
         lappend rootIcons "$i.$j"
      }
   }
}

#
# now that we've got our list of root icons, make up the hierarchy
# for each

foreach i $rootIcons { cwreg_MakeLinkHierarchy $i "" }
```

```
    #
    # the tidytree routine uses the root widget width as a basis for the
    # rest, so we need to make sure the widget is on-screen first

    update idletasks

    #
    # finally, for icons newly created, position them in a structured
    # tree format by using the icons' parent as the focus point

    foreach i $newWidgets {
       set s [cwreg_WidgetToStruct $i]
       set parent [$i cget -parent]
       if { $parent != "" } { $parent tidytree } else { $i tidytree }
    }

    #
    # To allow for receipt of events at a later date (e.g. requests to
    # join...) we look through any pending events and deal with them

    set events [collaborwriter keys cwevent]
    foreach i $events {
       cwreg_EnvironmentChanged "cwevent.$i"
    }

    #
    # Set up event catches for when the collaborwriter environment has
    # been added to (either a a new person has entered the world, or a
    # new team or project has been  created)

    collaborwriter bind addEnvInfo    [list cwreg_EnvironmentChanged %K]
    collaborwriter bind changeEnvInfo [list cwreg_EnvironmentChanged %K]
}

# ----------------------------------------------------------------------
# Start the client running
# ----------------------------------------------------------------------

gk_newenv -bind -client collaborwriter
gk_newenv localDesktop

localDesktop set dragging nodragging

cwreg_CreateView

#
# link to the central registrar

keylset info name [userprefs name]
set host [registrar host]
set fd [gk_connectToServer $host 9000 $info]
collaborwriter server $fd

# dp_RDO $fd gk_serverQueryID "cwreg_SetUniqueID"

gk_pollConferences

#
# initialise some information in the collaborwriter desktop fields. This
# will be overwritten if another conference exists (or the stuff is read
# from a file)

foreach type { proj team people } {
   localDesktop set desktop.nextIconPos.$type [list 20 40 0]
}

#
# The event handling mechanism has changed to use a binding mechanism
# as used in Tk gk_on and post_event commands are depreciated
```

```
#
# set up an event catch to get all the details from the central server -
# need to put a delay here otherwise we end up creating 2 versions of
# the same icon... ???

collaborwriter bind envReceived                                          \
   [list after 1000 cwreg_InitialiseSettings]

# ------------------------------------------------------------------------
# Set up all the registration policies at the end
# ------------------------------------------------------------------------

# Remove me from all of my conferences.

gk_on {[gk_event type]=="regClientQuitting"} {gk_killAllMyConfs}

# all new conferences are valid, and we add them to our list

gk_on {[gk_event type]=="foundNewConf"} {gk_addConfToList}

# when we find a new conference we created, add it to our current room;
# this will filter down and cause us to join to it

gk_on {([gk_event type]=="foundNewConf") &&                              \
   ([gk_event originator]==[gk_uniqprogid])} {
   gk_conferenceJoined [gk_event confnum]
   gk_callJoinConference [gk_event confnum]
   gk_pollUsers [gk_event confnum]

   cw_Debug "project node for conf is [gk_event projectNode]" green
   collaborwriter set [gk_event                                          \
      projectNode].joinedTools.[gk_event confnum] [gk_event confname]
}

# all new users are valid, and we add them to our list

gk_on {[gk_event type]=="foundNewUser"} {gk_addUserToList}

# when we find ourselves added to a conference, create the process

gk_on {  ([gk_event type]=="foundNewUser") &&                            \
         ([gk_event host]==[userprefs host]) &&                          \
         ([gk_event port]==[userprefs port])} {

   # confs c.[gk_event confnum].persistenceStyle no_persistence

   gk_createConference [gk_event confnum] [gk_event usernum]
   gk_joinToOthers [gk_event confnum] [gk_event usernum]
}

# if our user is removed from the conference we are no longer joined

gk_on {([gk_event type]=="foundDeletedUser") &&                          \
       ([gk_event usernum]==[spawned c.[gk_event confnum].localuser]) }\
   {gk_noLongerJoined [gk_event confnum]}

# all deleted users are removed from the lists and rooms

gk_on {[gk_event type]=="foundDeletedUser"} {
   gk_noLongerJoined [gk_event confnum];gk_removeUserFromList }

# when the last user of a conference leaves, we shut down the conference

gk_on {[gk_event type]=="lastUserLeftConf"} {
   gk_callDeleteConference [gk_event confnum]; gk_pollConferences}
```

```
gk_on {[gk_event type]=="conferenceDied"} {
   gk_noLongerJoined [gk_event conf]}

# when a conference connected to us died, report the user left

gk_on {[gk_event type]=="conferenceDied"} {
   gk_userLeft [gk_event conf] [gk_event user] [gk_event filedesc]}

# all deleted conferences are removed from lists
# all deleted conferences are removed from projects

gk_on {[gk_event type]=="foundDeletedConf"} {gk_removeConfFromList}

gk_on {[gk_event type]=="foundDeletedConf"} {
   set confNum [gk_event confnum]
   foreach node [collaborwriter keys proj] {
      set idx
         [lsearch [collaborwriter keys proj.$node.joinedTools] $confNum]
      if { $idx != -1 } {
         cw_Debug "deleting conf: proj.$node.joinedTools.$confNum" green
         collaborwriter delete proj.$node.joinedTools.$confNum
         break
      }
   }
}

# when a user asks to create a new conference, create it as requested

gk_on {[gk_event type]=="userRequestedNewConf"} {gk_createRequestedConf}

#
# Requests by the conference processes

gk_on {[gk_event type]=="editorRequestsInitialInfo"} {
   cwreg_SendInitialInfoToEditor [gk_event confnum] [gk_event id]
}

# ------------------------------------------------------------------------
# send the details of this world to a calling editor
# ------------------------------------------------------------------------

proc cwreg_SendInitialInfoToEditor { confnum id } {
   set allDetails [userprefs get collaborwriter.myDetails.$id]
   set node [lindex $allDetails 0]
   set me [lindex $allDetails 1]
   set p [cw_RemoveSpacesFromName [lindex [split $node .] 0] 1]

   #
   # get the list of members in the node

   set world {}
   set worldicons {}
   set worldcolours {}
   set peopleList [cwreg_GetMembersOf $node allNodes recurse]

   #
   # send the list of people and their respective icon definition (the
   # global variant) which is used for portraying their picture in the
   # editor tool

   foreach i $peopleList {
      lappend world [lindex $i 0]
      lappend worldicons [cwreg_FindIconPictureFor [lindex $i 0]]
      lappend worldcolours [collaborwriter get usercolour.[lindex $i 0]]
   }
```

```
    keylset event type sessmgrSendInfo confnum $confnum details $me      \
       world $world proj $p worldicons $worldicons                        \
       worldcolours $worldcolours
    gk_toConf $confnum "gk_postEvent \{$event\}"
}

# ---------------------------------------------------------------------
# As above, but send info whenever things change in the session mgr
# ---------------------------------------------------------------------

proc cwreg_SendUpdatedInfoToEditor { confnum node } {
    set world {}
    set worldicons {}
    set worldcolours {}

    set peopleList [cwreg_GetMembersOf $node allNodes recurse]
    foreach i $peopleList {
       lappend world [lindex $i 0]
       lappend worldicons [cwreg_FindIconPictureFor [lindex $i 0]]
       lappend worldcolours [collaborwriter get usercolour.[lindex $i 0]]
    }
    set p [cw_RemoveSpacesFromName [lindex [split $node .] 0] 1]

    keylset event type sessmgrSendInfo confnum $confnum details none     \
       world $world proj $p worldicons $worldicons                        \
       worldcolours $worldcolours
    gk_toConf $confnum "gk_postEvent \{$event\}"
}
```

Appendix C
Collaborwriter authoring tool

The code controls the main authoring environment, including authoring node creation and manipulation and comment creation.

C.1 Program code

```
#!/usr/local/bin/gkwish -f
# ------------------------------------------------------------------------
# editor.tool
# Text editor for Collaborwriter
#
# This follows on from the access rights assigned initially in the
# session manager. That is, a person has a set of initial rights
# (read comment write delegate) which determine what can do in a node.
#
# The rule policies adopted are:
#   Can create an icon  - if have AUTHOR or DELEGATE rights at ANY point
#   Can move an icon    - if have any rights for that node
#   Can link two icons  - if have AUTHOR or DELEGATE rights for both
#                         PARENT & CHILD
#   Can break link      - if have AUTHOR or DELEGATE rights for both
#                         PARENT & CHILD
#   Can type text node  - if have AUTHOR rights for that node
#   Can type comments   - if have AUTHOR or ANNOTATE rights for the node

# ------------------------------------------------------------------------
# The text editor consists of a node hierarchy forming a tree structure.
# Each node consists of a grouptable widget and a label. On clicking on
# the node, you enter text for that node. New nodes are created and
# manipulated in the same manner as used on the session manager layer.
# ------------------------------------------------------------------------

# ------------------------------------------------------------------------
# Initialise the conference process, linking it to the session manager
# ------------------------------------------------------------------------

source "[file dirname $argv0]/../library/envir.tcl"
source "[file dirname $argv0]/../library/conf.tcl"
gk_initConf $argv

# ------------------------------------------------------------------------
# Set up autoloading to use appropriate library routines plus overriding
# routines in this file
# ------------------------------------------------------------------------

set rootDir [file dirname $argv0]/../library
set auto_path [linsert $auto_path 0 "/usr/local/lib/groupkit"]
set auto_path [linsert $auto_path 0 $rootDir]
```

```
unset rootDir

# foreach dir $auto_path { source "$dir/tclIndex" }

users local.persistenceStyle always

# ------------------------------------------------------------------------
# Globals for the current selection/hilite (pointless as environment)
# ------------------------------------------------------------------------

set gNodeBinding 0
set gDraggingIcon 0
set gSelection ""
set gCurrentHilite ""
set gDefaultAbility UNDEFINED
set gProjectName ""

# ------------------------------------------------------------------------
# Fire off an event to get our initial abilities from the session mgr
# (ie. what we can do)
# ------------------------------------------------------------------------

if [gk_amOriginator] {
   keylset event type editorRequestsInitialInfo                          \
      confnum [users local.confnum] id [users local.originator]
} else {
   keylset event type editorRequestsInitialInfo                          \
      confnum [users local.confnum] id [users local.confnum]
}

cw_toRC $event

gk_on { ([gk_event type]=="sessmgrSendInfo") &&                          \
   ([gk_event confnum]==[users local.confnum]) } {

   if { [gk_event details] != "none" } {
      set gDefaultAbility [gk_event details]
   }

   textnodes set peopleInWorld [gk_event world]
   textnodes set peopleIcons [gk_event worldicons]
   textnodes set peopleColours [gk_event worldcolours]

   set gProjectName [gk_event proj]
   cw_Debug "Default ability: $gDefaultAbility" yellow
   cw_Debug "world: [gk_event world]" yellow

   set myIdx [lsearch [gk_event world] [users local.username]]
   users set local.iconpict [lindex [gk_event worldicons] $myIdx]

   set myColour [lindex [gk_event worldcolours] $myIdx]

   #
   # ??? need to change mycolour to the appropriate lighter shade

   SetCursorColourFor [users local.usernum] $myColour
}

# ------------------------------------------------------------------------
# Set up our colour in the gk user attributes
# ------------------------------------------------------------------------

proc SetCursorColourFor { usernum colour } {
   gk_setUserAttrib $usernum cursor_color $colour
}
```

```
proc SetCursorColourForOthers {} {
   gk_toOthers gk_setUserAttrib                                          \
      [users local.usernum] cursor_color [users local.cursor_color]
}

# ---------------------------------------------------------------------
# Routine required by multitext to return the colour of user <name>
# ---------------------------------------------------------------------

proc GetColourForUserName { name } {
   set name [cw_AddSpacesToName $name]
   set peopleInWorld [textnodes get peopleInWorld]
   set idx [lsearch $peopleInWorld $name]

   if { $idx == -1 } {
      cw_Debug "--- Couldn't find user $name to get colour ---" red
      return black
   }

   return [lindex [textnodes get peopleColours] $idx]
}

# ---------------------------------------------------------------------
# Find out what our ability is (returns a list of what we can do at a
# specific node)
# if node == "" then we return our default ability
# ---------------------------------------------------------------------

proc editor_GetAbility { node } {
   global gDefaultAbility
   if { $node == "" } { return $gDefaultAbility }

   set memberList [textnodes get node.$node.members]
   set myDetails                                                         \
      [editor_SearchMemberListFor $memberList [userprefs name]]

   #
   # if we arn't specified in a node, do we return {no no no no} or our
   # default ability - I reckon { no no no no } - POLICY DECISION ???

   if { $myDetails == -1 } { set myDetails [list blah no no no no] }

   return [lrange $myDetails 1 end]
}

# ---------------------------------------------------------------------
# Utility routine to get a boolean from a yes/no/inherit
# ---------------------------------------------------------------------

proc IsYes { thing } {
   if { $thing == "yes" } { return 1 }
   return 0
}

# ---------------------------------------------------------------------
# Find out if we have authoring rights in ANY node (affects whether we
# can create an icon)
# That means writing or delegation rights, by the way
# ---------------------------------------------------------------------

set gCanAuthorAtAll 0

proc _editor_CanAuthorAtAll { node } {
   global gCanAuthorAtAll

   if { $gCanAuthorAtAll == 0 } {
      set memberList [textnodes get node.$node.members]
```

```
      set myInfo
         [editor_SearchMemberListFor $memberList [userprefs name]]
      if { $myInfo != -1 } {
         set gCanAuthorAtAll [expr
            ([IsYes [lindex $myInfo 3]] || [IsYes [lindex $myInfo 4]])]
      }
   }
}

proc editor_CanAuthorAtAll {} {
   global gCanAuthorAtAll gDefaultAbility
   set gCanAuthorAtAll [expr ([IsYes [lindex $gDefaultAbility 2]] ||
      [IsYes [lindex $gDefaultAbility 3]]) ]
   if { $gCanAuthorAtAll == 0 } {
      texttree dotoall tree _editor_CanAuthorAtAll ? }

   return $gCanAuthorAtAll
}

# ---------------------------------------------------------------------
# Utility routines returning true/false for ability to
# read/comment/write/delegate
# ---------------------------------------------------------------------

proc editor_CanRead { node } {
   set abilities [editor_GetAbility $node]
   return [IsYes [lindex $abilities 0]]
}

proc editor_CanComment { node } {
   set abilities [editor_GetAbility $node]
   return [IsYes [lindex $abilities 1]]
}

proc editor_CanWrite { node } {
   set abilities [editor_GetAbility $node]
   return [IsYes [lindex $abilities 2]]
}

proc editor_CanDelegate { node } {
   set abilities [editor_GetAbility $node]
   return [IsYes [lindex $abilities 3]]
}

proc editor_CanManipulate { node } {
   set abilities [editor_GetAbility $node]
   return [expr
      [IsYes [lindex $abilities 2]] || [IsYes [lindex $abilities 3]]]
}

# ---------------------------------------------------------------------
# Change the display font of the text widget.
# Based on code in the text editor developed by Michael Tauber
# ---------------------------------------------------------------------

set font "*-Times-medium-r-*-"
set fontsize 12
set nfont $font
set fsize $fontsize

proc editor_ChangeEditorFont { w } {
   global font fontsize nfont fsize

   if {[catch [list $w.f2.editor configure -font "$nfont$fsize-*"]]>0} {
      catch {destroy .fehl}
```

```
        toplevel .fehl
        wm geometry .fehl +300+250
        wm title .fehl "ERROR !!!"
        label .fehl.meld -text "Font $nfont$fsize does not exist"
        button .fehl.ok -text "OK" -relief raised -command "destroy .fehl"
        pack .fehl.meld -side top -fill x
        pack .fehl.ok -side bottom -fill x -expand yes
    } else {
        set font $nfont
        set fontsize $fsize
        destroy .font
    }
}

proc editor_ChgFont { w } {

    global font fontsize nfont fsize

    toplevel .font
    label .font.l -text "Current Font: $font"
    frame .font.f
    listbox .font.fontlist -relief sunken -width 24 -height 8
    .font.fontlist insert end Courier Helvetica Fixed                    \
      "New Century Schoolbook" Times Lucida Lucidabright Lucidatypewriter
    listbox .font.fontsize -relief sunken -width 4 -height 6
    .font.fontsize insert end  8 10 12 14 18 24

    label .font.f.bsp -text "Example Text" -relief ridge                 \
       -font "$font$fontsize-*"
    frame .font.f2
    button .font.f2.nok -text "Abort" -command {destroy .font}
    button .font.f2.ok -text "OK"                                        \
       -command [list editor_ChangeEditorFont $w]

    pack .font.l
    pack .font.f.bsp -fill x -padx 5 -pady 5
    pack .font.f2.ok .font.f2.nok -fill x -side bottom -pady 5 -ipadx 2
    pack .font.fontlist .font.fontsize .font.f .font.f2                  \
       -side left -padx 5 -pady 5
    bind .font.fontlist <Double-Button-1> {
       set i [.font.fontlist curselection]
       switch $i {
          0 {set nfont *-Courier-medium-r-*-}
          1 {set nfont *-Helvetica-medium-r-*-}
           2 {set nfont *-fixed-medium-r-*-}
           3 {set nfont "*-New Century Schoolbook-medium-r-*-"}
           4 {set nfont *-Times-medium-r-*-}
           5 {set nfont *-Lucida-medium-r-*-}
           6 {set nfont *-Lucidabright-medium-r-*-}
           7 {set nfont *-Lucidatypewriter-medium-r-*-}
       }
       if { [catch {.font.f.bsp config -font "$nfont$fsize-*"}] > 0 } {
           catch {destroy .fehl}
           toplevel .fehl
           wm geometry .fehl +300+250
           wm title .fehl "ERROR !!!"
           label .fehl.meld -text "The font $nfont$fsize does not exist"
           button .fehl.ok -text "OK" -relief raised                     \
              -command "destroy .fehl"
           pack .fehl.meld -side top -fill x
           pack .fehl.ok -side bottom -fill x -expand yes
       }
    }
```

```
    bind .font.fontsize <Double-Button-1> {
        set index2 [.font.fontsize curselection]
        switch $index2 {
            0 {set fsize 8}
            1 {set fsize 10}
            2 {set fsize 12}
            3 {set fsize 14}
            4 {set fsize 18}
            5 {set fsize 24}
        }
        if { [catch {.font.f.bsp config -font "$nfont$fsize-*"}] > 0 } {
            catch {destroy .fehl}
            toplevel .fehl
            wm geometry .fehl +300+250
            wm title .fehl "ERROR !!!"
            label .fehl.meld -text "This font does not exist"
            button .fehl.ok -text "OK" -relief raised                    \
                -command "destroy .fehl"
            pack .fehl.meld -side top -fill x
            pack .fehl.ok -side bottom -fill x -expand yes
        }
    }
}

# ---------------------------------------------------------------------
# Create the display for when we enter a node
# ---------------------------------------------------------------------

proc editor_EnterNode { w node nodeID } {
    global gProjectName

    if { [winfo exists $w] } { raise $w; return }

    #
    # if we don't have read permission we can't read what's in this node

    if ![editor_CanRead $node] { return }

    toplevel $w

    wm title $w [editor_GetNodeTextName $node force]
    set id [cw_ChangeDotInName $w]
    set env "$gProjectName$node"
    set me [users local.usernum]

    #
    # first we see if there's anybody in this node

    set currentPeople [textnodes get node$w.currentPeople]
    set idx [cw_ListSearch $currentPeople 0 $me]
    if { $idx == -1 } {
        lappend currentPeople [list $me [userprefs color]]
        textnodes set node$w.currentPeople $currentPeople
    }

    gk_toAll ".c$w.icon" adduser $me [userprefs color]

    frame $w.f -bd 4 -relief raised
    frame $w.f2; frame $w.f.lhs; frame $w.f.rhs

    attrpicturebar $w.f.lhs.picturebar -scrollid $id                  \
        -canchange [editor_CanDelegate $node]                        \
        -command "cw_NotImplementedYet"
    gk_scrollbar $w.f2.sb -command "$w.f2.editor.t yview"            \
        -repeatdelay 1000 -scrollid $id
```

```
    multitext $w.f2.editor -height 15 -width 60 -wrap word                \
       -setgrid true -coupling none -yscrollcommand [list $w.f2.sb set] \
       -scrollid $id -spacing2 2 -spacing3 12                             \
       -font "*-Times-medium-r-*-12-*" -envname $env -coupling tight      \
       -bg FloralWhite -showcolour 1

    wm protocol $w WM_DELETE_WINDOW "editor_LeaveNode $w $id $node $env"

    gk_defaultMenu $w.mbar

    $w.mbar.file.menu entryconfigure 1 -label Close                       \
       -command "editor_LeaveNode $w $id $node $env"
    $w.mbar add viewmenu 1 -text View
    $w.mbar add commentmenu 2 -text Comments
    $w.mbar itemcommand 0 add command -label "Revert to last session"
    $w.mbar itemcommand 1 add command -label "Change font..."             \
       -command "editor_ChgFont $w"
    $w.mbar itemcommand 1 add checkbutton -label "Coloured text"          \
       -variable "$w.f2.editor\(-showcolour\)"                            \
       -command "editor_ToggleMultitextColour $w.f2.editor"
    $w.mbar itemcommand 1 add cascade -label "Show comments of"           \
       -menu $w.mbar.viewmenu.menu.m
    $w.mbar itemcommand 2 add command -label "Add comment"                \
       -command "editor_AddComment $w $node $w.f2.editor"
    $w.mbar itemcommand 2 add command -label "Remove comment"             \
       -command "editor_RemoveComment $w $node $w.f2.editor"

    menu $w.mbar.viewmenu.menu.m -tearoff 0 -postcommand                  \
       [list editor_ViewCommentsOfMenu $w.mbar.viewmenu.menu.m $node]

    if { [llength $currentPeople] == 1 } {

       #
       # we're the only person in the node, so see if it has a text file
       # associated with it
       # THIS SHOULD BE RETURNED AS A REQUEST FROM THE SESSION MANAGER,
       # WHICH IN TURN ASKS THE REGISTRAR (THUS ONLY 1 COPY OF DATA) ???

       $w.f2.editor initialise
    } else {

       #
       # there are others about, so we ask them for their details.
       # We're appended to the end of the list, so just ask the first
       # person

       gk_toUserNum [lindex [lindex $currentPeople 0] 0]                  \
          multitext_SendDetailsTo $w.f2.editor $me
    }

    if ![editor_CanWrite $node] {$w.f2.editor configure -state disabled}

    pack $w.mbar -side top -fill x
    pack $w.f -side top -fill x
    pack $w.f2 -fill both -expand yes
    pack $w.f.rhs -side right
    pack $w.f.lhs -side left -fill x -expand yes
    pack $w.f.lhs.picturebar -side left -fill x
    pack $w.f2.sb -side right -fill y
    pack $w.f2.editor -side left -fill both -expand yes
}

# ---------------------------------------------------------------------
# Tidy up everything on exit from the node
# ---------------------------------------------------------------------
```

```
proc editor_LeaveNode { w id node env } {
    set currentPeople [textnodes get node$w.currentPeople]
    set idx [cw_ListSearch $currentPeople 0 [users local.usernum]]

    if { $idx != -1 } { textnodes set node$w.currentPeople                    \
        [lreplace $currentPeople $idx $idx]
    }

    #
    # tell the groupnode that we're not editing it anymore - only need
    # to do this for the top level - views at the window node are
    # are automatically removed

    gk_toAll ".c$w.icon" deleteuser direct [users local.usernum]
    cw_Debug "leaving node: $w - id: $id" red

    #
    # if I'm the last user then I save the text to a file...

    if { [llength $currentPeople] == 1 } {
        $w.f2.editor save
    }

    gk_viewsDelete $id [users local.usernum]
    destroy $w
}

# ---------------------------------------------------------------------
# Set the colour of the multitext display to colour or b&w
# ---------------------------------------------------------------------

proc editor_ToggleMultitextColour { w } {
    set state [$w cget -showcolour]
    $w configure -showcolour $state
}

# ---------------------------------------------------------------------
# Insert some text - depreciated routine
# ---------------------------------------------------------------------

proc editor_InsertBlockOfText { w txt } {
    cw_Debug "** insert block of text $txt **" white
    $w.f2.editor insert 0.1 "$txt"
}

# ---------------------------------------------------------------------
# Build a menu showing who has created comments - AND WHEN (user reqst)
# ---------------------------------------------------------------------

proc editor_ViewCommentsOfMenu { m node } {
    $m delete 0 end
    $m add command -label None
    $m add command -label All
    $m add separator

    # ??? NIY: List of people who have created comments
}

# ---------------------------------------------------------------------
# Add a comment to the text
# ---------------------------------------------------------------------

proc editor_AddComment { window node w } {
    if ![editor_CanComment $node] { return }
    set sel [$w tag ranges sel]
    set wholeNode 0
```

```
    #
    # if there's no selection then the comment applies to all the text
    # in the node

    if { $sel == "" } {
        $w tag add sel 1.0 [$w index end]
        set sel [$w tag ranges sel]
        set wholeNode 1
    }

    #
    # pull out the text from the selection

    set commentText ""
    set items [expr ([llength $sel] / 2)]
    for { set i 0 } { $i < $items } { incr i } {
        set idx [expr $i * 2]
        set m [$w get [lindex $sel $idx] [lindex $sel [expr $idx +1]]]
        set commentText "$commentText$m"
    }

    if { [string length $commentText] > 180 } {
        set commentText "[string range $commentText 0 160]..."
        if { $wholeNode == 1 } {
            set commentText "$commentText plus the rest of the node."
        }
    }

    #
    # pop up a comment window (we do a forced grab here for simplicity
    # sake...)

    set cw "$w.p[users local.usernum]"
    toplevel $cw
    wm title $cw "Add comment"
    set gComment "gCommentType$cw"
    eval "global $gComment"
    eval "set $gComment informative"

    frame $cw.f
    eval "radiobutton $cw.f.inform -text Informative                    \
        -variable $gComment -value informative"
    eval "radiobutton $cw.f.reserved -text Reserved                     \
        -variable $gComment -value reserved"
    label $cw.l -text "The original text is:"                           \
        -font *-Times-medium-r-*-10-*
    message $cw.m -text "$commentText" -width 380 -bd 1 -relief groove   \
        -font *-Times-medium-r-*-12-*
    text $cw.t -background yellow -foreground black -width 60 -height 6
                                                                        \
    button $cw.ok -text "OK"                                            \
        -command [list editor_CompleteComment $w $node $cw $sel 1]
    button $cw.cancel -text "Cancel"                                    \
        -command [list editor_CompleteComment $w $node $cw $sel 0]

    pack $cw.f -fill x -expand yes
    pack $cw.l -anchor w
    pack $cw.m $cw.t -fill x -expand yes
    pack $cw.f.inform $cw.f.reserved -side left -expand yes
    pack $cw.cancel $cw.ok -side left -expand yes
    tkwait visibility $cw
    # grab set -global $cw
}
```

```
# ------------------------------------------------------------------------
# Routine required by multitext to return the colour of user <name>
# ------------------------------------------------------------------------

proc editor_CompleteComment { w node cw sel ok } {

    #
    # we add a comment to the selection - the is currently implemented
    # by a selection - there's positioning problems when we include a
    # marker character (like we had with remote cursors) ????

    set gComment "gCommentType$cw"
    eval "global $gComment"

    set txt [$cw.t get 1.0 [$cw.t index end]]
    destroy $cw
    if { $ok == 0 } { return }

    set commentID "comment_[cw_GetCounter]"
    eval "$w tag add $commentID $sel"
    $w tag configure $commentID -underline 1
    $w tag bind $commentID <Double-Button-2>                              \
       "editor_ShowComments $node $commentID $w"
    set comments [textnodes get node.$node.comments]

    #
    # if the comment is a reserved one set the bindings to trigger the
    # negotiation tool ???? NIY

    if { [eval "set $gComment"] == "reserved" } {
       cw_NotImplementedYet "reserved comment link"
    }
    lappend comments [list $commentID $txt [userprefs name] $sel]
    textnodes set node.$node.comments $comments
}

# ------------------------------------------------------------------------
# Show the comment for a thing
# ------------------------------------------------------------------------

proc editor_ShowComments { node commentID w } {
    set comments [textnodes get node.$node.comments]
    set idx [cw_ListSearch $comments 0 $commentID]
    set myComment [lindex $comments $idx]
    cw_Debug "[lindex $myComment 1]" green
}

# ------------------------------------------------------------------------
# Utility routines for converting widget to node id and vice-versa
# ------------------------------------------------------------------------

proc editor_WidgetToNode { w } {
    set w [string range $w 3 end]
    set pos [string first "." $w]
    if { $pos != -1 } { set w [string range $w 0 [expr $pos - 1]] }
    return $w
}

proc editor_NodeToWidget { w } {
    return ".c.$w"
}

# ------------------------------------------------------------------------
# Node drag, select and link creation/deletion routines
# ------------------------------------------------------------------------
```

```
proc editor_NewXY { nodeW x y } {
   set node [editor_WidgetToNode $nodeW]
   textnodes set node.$node.x $x
   textnodes set node.$node.y $y
}

proc editor_BreakLink { nodeW parentW } {
   set parent [editor_WidgetToNode $parentW]
   set child [editor_WidgetToNode $nodeW]

   #
   # we can only break a link between nodes if we have author or editor
   # access to BOTH

   if {[editor_CanManipulate $parent] && [editor_CanManipulate $child]}{
      textnodes set node.$child.parent ""
      set coords [texttree getposition $child]
      set parentpath [texttree getparent $child]
      editor_NodeHierarchyChanged "" $child $coords $parentpath
   }
}

proc editor_StartDragIconImage { w } {
   global gSelection gDraggingIcon gCurrentHilite

   if { $gCurrentHilite != "" } { $gCurrentHilite highlight reset all }
   editor_ClearSelection
   set gSelection $w
   set gDraggingIcon 0
   set gCurrentHilite ""

   $w highlight yellow all
}

proc editor_ClearSelection {} {
   global gSelection

   if { $gSelection != "" } {
      $gSelection highlight reset all
      set gSelection ""
   }
}

proc editor_StopDragIconImage { startW x y } {
   global gDraggingIcon gCurrentHilite

   set endW [editor_GetWidgetUnderMouse $startW $x $y]

   #
   # if we've been dragging the icon, lose the selection we created

   if { $gDraggingIcon == 1 } {
      . configure -cursor {}
      editor_ClearSelection
   }

   if { $gCurrentHilite != "" } {
      $gCurrentHilite highlight reset all
      set gCurrentHilite ""
   }

   if { $endW != "" } {
      set parent [editor_WidgetToNode $endW]
      set child [editor_WidgetToNode $startW]

      #
      # we only add a link between nodes if we have author or editor
      # access to BOTH
```

```
      if {[editor_CanManipulate $parent] &&                                \
                                          [editor_CanManipulate $child]} {
         textnodes set node.$child.parent                                  \
            [textnodes get node.$parent.id]

         #
         # add this node to our texttree hierarchical structure

         set coords [texttree getposition $child]
         set parentpath [texttree getparent $child]
         editor_NodeHierarchyChanged $parent $child $coords $parentpath
      }
   }
}

proc editor_DragIconImage { startW x y } {
   global gDraggingIcon gCurrentHilite

   if { $gDraggingIcon == 0 } {
      set gDraggingIcon 1
      . configure -cursor [list @bitmaps/cursornode black]
   }

   set w [editor_GetWidgetUnderMouse $startW $x $y]
   if { $w != $gCurrentHilite } {
      if { $gCurrentHilite != "" } {$gCurrentHilite highlight reset all}
      set gCurrentHilite $w
      if { $gCurrentHilite != "" } {$gCurrentHilite highlight cyan all}
   }
}

proc editor_GetWidgetUnderMouse { startW x y } {
   set node [editor_WidgetToNode $startW]
   if { [textnodes get node.$node.parent] != "" } { return "" }

   set w [winfo containing $x $y]
   if { [string first ".c." $w] != 0 } { return "" }

   set w [string range $w 3 end]
   set pos [string first "." $w]
   if { $pos != -1 } { set w [string range $w 0 [expr $pos - 1]] }
   set w ".c.$w"

   set root [$w getroot]
   if { [$root ismember $startW] == 1 } { return "" }
   return $w
}

# ------------------------------------------------------------------------
# Automagic node naming
# ------------------------------------------------------------------------

proc editor_NodeHierarchyChanged { parent child coords parentpath } {

   #
   # get the autonaming updated as appropriate

   texttree addbelow $parent $child
   texttree dotoall $child editor_GetNodeTextName ? force

   set nodesToUpdate [cw_ReverseList [texttree keys $parentpath]]
   set idx [expr [lindex $coords 1] -2]
   if { $idx >= 0 } {set nodesToUpdate [lreplace $nodesToUpdate 0 $idx]}
   foreach i $nodesToUpdate {
      texttree dotoall $i editor_GetNodeTextName ? force
   }
}
```

```
proc editor_GetNodeTextName { node reevaluate } {
   if { ![textnodes get node.$node.autonumber] } {
      return [textnodes get node.$node.text]
   }

   if {[textnodes get node.$node.valueset]&& ($reevaluate != "force")} {
      return [textnodes get node.$node.text]
   }

   #
   # we will be creating the node name here, so flag this fact already

   textnodes set node.$node.valueset 1

   set parentNode [textnodes get node.$node.parent]
   if { $parentNode == "" } {
      set structurenum ""
   } else {
      set structurenum "[textnodes get node.$parentNode.sectionlevel]."
   }

   set coords [texttree getposition $node]
   set sectionlevel "$structurenum[lindex $coords 1]"
   set txt "Section $sectionlevel"
   textnodes set node.$node.text $txt
   textnodes set node.$node.sectionlevel $sectionlevel

   return $txt
}

# ---------------------------------------------------------------------
# Update the screen display rebuilding nodes as necessary
# ---------------------------------------------------------------------

proc editor_UpdateEnvironment {} {

   #
   # update the node data structures

   foreach node [textnodes keys node] {
      textnodes set node.$node.id [textnodes get node.$node.id]
   }

   #
   # get the links built up at the end

   foreach node [textnodes keys node] {
      if { [textnodes get node.$node.parent] != "" } {
         textnodes set node.$node.parent                    \
            [textnodes get node.$node.parent]
      }
   }
}

# ---------------------------------------------------------------------
# The author node attributes for a node - same as used in session
# manager project attributes
# ---------------------------------------------------------------------

proc editor_NodeAttributes { w } {
   set dlog ".mnd_[cw_ChangeDotInName $w]"
   if ![cw_CreateDialog $dlog] { return }

   set node [editor_WidgetToNode $w]

   global "{$dlog}mvars"
   set "{$dlog}mvars(node)" $node
```

```
set "{$dlog}mvars(listSize)" 0
set "{$dlog}mvars(listIndex)" 0
set "{$dlog}mvars(removeList)" {}
set labelName [editor_GetNodeTextName $node dontreevaluate]

wm title $dlog "Modify text node \"$labelName\""

frame $dlog.f -bd 4 -relief ridge
frame $dlog.f.n; frame $dlog.f.p; frame $dlog.f.j; frame $dlog.f.r
frame $dlog.f.s
label $dlog.f.n.l -pady 0 -bd 0 -text "\nProject member"
label $dlog.f.p.l -pady 0 -bd 0                                          \
   -text "          ------ Initial Access Rights ------"
label $dlog.f.p.l2 -pady 3 -bd 0                                         \
   -text "      Read       Comment       Write        Delegate"
label $dlog.f.s.l -pady 0 -bd 0 -text "\n"
listbox $dlog.f.n.namelist -width 20 -selectmode single                  \
   -yscrollcommand [list boxscroll_ListChanged "{$dlog}mvars"            \
   $dlog $dlog.f.s.scrV set]
scrollbar $dlog.f.s.scrV -orient vertical -command [list                 \
boxscroll_ListChanged "{$dlog}mvars" $dlog $dlog.f.n.namelist yview]

frame $dlog.l
menubutton $dlog.l.adduser -text "Add person" -indicatoron 1             \
   -menu $dlog.l.adduser.m -relief raised -bd 2

#
# making this menu not a -tearoff doesn't seem to work ???

menu $dlog.l.adduser.m -postcommand [list editor_ChangeNodeMenu          \
   "{$dlog}mvars" $dlog.l.adduser.m $node $dlog]

button $dlog.l.defaultuser -text "Default attributes"                    \
   -state disabled                                                       \
   -command [list editor_SetToDefaultAttrs "{$dlog}mvars" $dlog]
button $dlog.l.removeuser -text "Remove person" -state disabled          \
   -command [list editor_DeleteUserFromNode "{$dlog}mvars" $dlog]

#
# modify the default widgets

$dlog.ok configure -text OK                                              \
   -command [list editor_ModifyNodeOK "{$dlog}mvars" $dlog]
$dlog.cancel configure                                                   \
   -command [list editor_ModifyNodeCancel "{$dlog}mvars" $dlog]

#
# set up the popup menu texts as used in the dialog

set "{$dlog}mvars(menutable.1.items)" [list yes no]
set "{$dlog}mvars(menutable.2.items)" [list yes no]
set "{$dlog}mvars(menutable.3.items)" [list yes no]
set "{$dlog}mvars(menutable.4.items)" [list yes no]

for { set i 1 } { $i < 5 } { incr i } {
   set "{$dlog}mvars(menutable.$i.cmd)" "editor_ChgBtnVal"
}

editor_ReadNodeEnv $dlog "{$dlog}mvars" ""

#
# get the dialog placed on the screen

pack $dlog.f $dlog.l -fill x
pack $dlog.f.s $dlog.f.p -side right -fill y
pack $dlog.f.n -fill both
```

```
    pack $dlog.f.n.l $dlog.f.p.l $dlog.f.s.l -side top -anchor w
    pack $dlog.f.p.l2 -side top -anchor w

    boxscroll_UpdateDialog "{$dlog}mvars" $dlog

    pack $dlog.f.n.namelist -fill both
    pack $dlog.f.s.scrV -side right -fill y
    pack $dlog.l.adduser $dlog.l.defaultuser $dlog.l.removeuser          \
       -side left -padx 10 -pady 10 -expand yes
    pack $dlog.cancel -side left -padx 10 -pady 10
    pack $dlog.ok -side right -padx 10 -pady 10

    #
    # set up any necessary bindings for the dialog

    set "{$dlog}mvars(oldBind)"                                          \
       "editor_ListboxBtn $dlog.f.n.namelist $dlog.l $dlog"
    bind Listbox <Button-1>                                              \
       +[list editor_ListboxBtn $dlog.f.n.namelist $dlog.l $dlog]

    set "{$dlog}mvars(CW1bind)" [textnodes bind addEnvInfo               \
       [list editor_ModifyNodeEnvChanged $dlog $node %K]]
    set "{$dlog}mvars(CW2bind)" [textnodes bind changeEnvInfo            \
       [list editor_ModifyNodeEnvChanged $dlog $node %K]]
}

# ---------------------------------------------------------------------
# Link for highlighting of remove user / default attrs buttons on the
# attributes dialog
# ---------------------------------------------------------------------

proc editor_ListboxBtn { linkedList dlog v } {
    set vars "{$v}mvars"
    upvar #0 $vars mvars

    if { $dlog != "" } {
       set sel [$linkedList curselection]
       if { $sel == "" } { set state disabled
       } else { set state normal }
       catch { $dlog.removeuser configure -state $state }

       if { $sel != "" } {
          if { $mvars(typeArray$sel) != "local" } { set state disabled }
       }
       catch { $dlog.defaultuser configure -state $state }
    }
}

# ---------------------------------------------------------------------
# Routine to change a button value by changing its linked array
# ---------------------------------------------------------------------

proc editor_ChgBtnVal { vars col row val w fromMenu } {
    upvar #0 $vars mvars
    set mvars(cbArray$col-$row) $val
    set mvars(stateArray$col-[expr $mvars(listIndex) + $row]) $val

    if { ($fromMenu == 1) &&                                             \
       ($mvars(typeArray[expr $mvars(listIndex) + $row])=="inherit")} {
       set mvars(typeArray[expr $mvars(listIndex) + $row]) local

       for { set j 1 } { $j < $mvars(numAttrs) } { incr j } {
          catch {$w.cb$j-$row configure -fg red}
       }
    }
}
```

```
# ------------------------------------------------------------------------
# Build the menu listing those people in the world who ARN'T in this
# node
# ------------------------------------------------------------------------

proc editor_ChangeNodeMenu { vars m node dlog } {
   upvar #0 $vars mvars

   set peopleInWorld [textnodes get peopleInWorld]
   set peopleInNode $mvars(memberList)
   $m delete 0 last
   $m configure -tearoff 0

   foreach name $peopleInWorld {
      if { [editor_SearchMemberListFor $peopleInNode $name] == -1 } {
         $m add command -label "$name" -command                          \
            [list editor_AddUserToNode $vars $node $name $dlog]
      }
   }
}

# ------------------------------------------------------------------------
# Transfer a user from the popup menu to the node
# (default attrs are read-only)
# ------------------------------------------------------------------------

proc editor_AddUserToNode { vars node username dlog } {
   upvar #0 $vars mvars
   lappend mvars(memberList) [list [list $username local] yes no no no]

   editor_ReadNodeEnv $dlog $vars $mvars(memberList)
   boxscroll_UpdateDialog $vars $dlog
}

proc editor_DeleteUserFromNode { vars dlog } {
   upvar #0 $vars mvars

   set sel [$dlog.f.n.namelist curselection]
   if { ($sel != "") } {
      set valuesToNo 0

      if { ($mvars(typeArray$sel) == "local") } {

         #
         # we've got a local thing here, so see if it has been
         # inherited from higher

         set parents [editor_GetCompleteMemberList                       \
            [textnodes get node.$mvars(node).parent]]
         set details [editor_SearchMemberListFor $parents                \
            $mvars(nameArray$sel)]

        if { $details == -1 } {

           #
           # the item was created at this point, so delete the item
           # completely

           set me [editor_SearchMemberListFor $mvars(memberList)         \
              $mvars(nameArray$sel)]
           lappend mvars(removeList) $mvars(nameArray$sel)
           set idx [lsearch $mvars(memberList) $me]
           set mvars(memberList)                                         \
              [lreplace $mvars(memberList) $idx $idx]
           editor_ReadNodeEnv $dlog $vars $mvars(memberList)
           boxscroll_UpdateDialog $vars $dlog
```

```
            } else {
              set valuesToNo 1
            }
        } else {
            set valuesToNo 1
        }

        if { $valuesToNo == 1 } {

            #
            # the item is inherited, so make it local with {no no no no}
            # settings

            set mvars(typeArray$sel) "local"
            set colour red
            for { set j 1 } { $j < $mvars(numAttrs) } { incr j } {
               editor_ChgBtnVal $vars $j $sel no $dlog.f.p.f$sel 0
               $dlog.f.p.f$sel.cb$j-$sel configure -fg $colour
            }
        }
    }
}

# ---------------------------------------------------------------------
# Set the selected user to their parent setting. If there is no parent
# then uses default attrs
# ---------------------------------------------------------------------

proc editor_SetToDefaultAttrs { vars dlog } {
   upvar #0 $vars mvars

   set sel [$dlog.f.n.namelist curselection]
   if { ($sel != "") && ($mvars(typeArray$sel) == "local") } {

      #
      # we've got a local thing here, so see if it has been inherited
      # from higher

      set parents [editor_GetCompleteMemberList                    \
         [textnodes get node.$mvars(node).parent]]
      set details [editor_SearchMemberListFor $parents             \
         $mvars(nameArray$sel)]

      if { $details == -1 } {

          #
          # the item was created at this point, so just give it
          # default settings

          set newDetails [list yes no no no]
          set colour red
      } else {

          #
          # the item has a parent, so make it inherited again with
          # those settings

          set newDetails [lrange $details 1 end]
          set mvars(typeArray$sel) "inherit"
          set colour black
      }

      for { set j 1 } { $j < $mvars(numAttrs) } { incr j } {
          editor_ChgBtnVal $vars $j $sel                           \
             [lindex $newDetails [expr $j-1]] $dlog.f.p.f$sel 0
          $dlog.f.p.f$sel.cb$j-$sel configure -fg $colour
```

```
        }
    }
}

# ------------------------------------------------------------------------
# Add the details changed in the node to the textnodes environment
# structure
# ------------------------------------------------------------------------

proc editor_ModifyNodeOK { vars dlog } {
    upvar #0 $vars mvars

    #
    # first update the fields detailing the member attributes

    set textnodesChanged 0
    set origList [textnodes get node.$mvars(node).members]

    for { set i 0 } { $i < $mvars(totalMembers) } { incr i } {
        if { $mvars(typeArray$i) == "inherit" } { continue }
        set origInfo [editor_SearchMemberListFor $origList                    \
            $mvars(nameArray$i)]
        set thisItemChanged 0

        set newItem [list $mvars(nameArray$i) local]
        set details {}
        for { set j 1 } { $j < $mvars(numAttrs) } { incr j } {
            if {($origInfo != -1) &&                                          \
                ($mvars(stateArray$j-$i) != [lindex $origInfo $j])} {
                set thisItemChanged 1
            }
            lappend details $mvars(stateArray$j-$i)
        }

        set newItem [concat [list $newItem] $details]
        if { $origInfo == -1 } {
            lappend origList $newItem
            set textnodesChanged 1
        } elseif { $thisItemChanged == 1 } {
            set idx [lsearch $origList $origInfo]
            set origList [lreplace $origList $idx $idx $newItem]
            set textnodesChanged 1
        }
    }

    #
    # see if any items were deleted from the member list

    foreach i $mvars(removeList) {
        set details [editor_SearchMemberListFor $origList $i]
        if { $details != -1 } {
            set idx [lsearch $origList $details]
            if { $idx != -1 } {
                set origList [lreplace $origList $idx $idx]
                set textnodesChanged 1
            }
        }
    }

    if { $textnodesChanged == 1 } {
        textnodes set node.$mvars(node).members $origList
    }

    editor_ModifyNodeCancel $vars $dlog
}
```

```
proc editor_ModifyNodeCancel { vars dlog } {
   upvar #0 $vars mvars

   textnodes delbind $mvars(CW1bind)
   textnodes delbind $mvars(CW2bind)
   set binding [bind Listbox <Button-1>]
   regsub $mvars(oldBind) $binding "" binding
   bind Listbox <Button-1> $binding

   unset mvars
   cw_RemoveDialog $dlog
}

# ----------------------------------------------------------------------
# Environment change while displaying the modify... dialog
# ----------------------------------------------------------------------

proc editor_ModifyNodeEnvChanged { dlog node key } {
   set parts [split $key .]
   set lastPart [lindex $parts end]
   set relevantNodes [concat [texttree getparents $node] $node]

   if {($lastPart == "members") &&                                    \
      ([lsearch $relevantNodes [lindex $parts 1]] != -1)} {

      #
      # our membership list has changed for this particular object

      editor_ReadNodeEnv $dlog "{$dlog}mvars" ""
      boxscroll_UpdateDialog "{$dlog}mvars" $dlog
   }
}

# ----------------------------------------------------------------------
# Read in the environment details for use in the node dialogs
# ----------------------------------------------------------------------

proc editor_ReadNodeEnv { dlog vars memberList } {
   upvar #0 $vars mvars

   set node $mvars(node)
   if { $memberList == "" } {
      set memberList [editor_GetCompleteMemberList $node]
   }
   set numAttrs [llength [lindex $memberList 0]]

   set totalMembers 0
   $dlog.f.n.namelist delete 0 end

   foreach i $memberList {
      set item [lindex $i 0]
      set name [lindex $item 0]
      if { [lsearch $mvars(removeList) $name] == -1 } {
         $dlog.f.n.namelist insert end $name
         set mvars(nameArray$totalMembers) $name
         for { set j 1 } { $j < $numAttrs } { incr j } {
            set mvars(stateArray$j-$totalMembers) [lindex $i $j]
         }
         set mvars(typeArray$totalMembers) [lindex $item 1]

         incr totalMembers
      }
   }

   set mvars(totalMembers) $totalMembers
   set mvars(memberList) $memberList
```

```
    set mvars(numAttrs) $numAttrs
}

# ---------------------------------------------------------------------
# USEFUL ROUTINE
# Search an editor 'memberlist' for a particular username
# Returns -1 if not found, otherwise that user's details
# ---------------------------------------------------------------------

proc editor_SearchMemberListFor { memberList name } {
    set found -1
    foreach i $memberList {
        if { [lindex [lindex $i 0] 0] == $name } { set found $i; break }
    }
    return $found
}

# ---------------------------------------------------------------------
# Return a list of all the people who has access to this node
# People higher up the hierarchy are given 'inherited' status, and are
# shown with the status set at their lowest level
# (eg. reader->writer :writer)
# ---------------------------------------------------------------------

proc editor_GetCompleteMemberList { baseNode } {
    set memberList [textnodes get node.$baseNode.members]

    set parentNodes [cw_ReverseList [texttree getparents $baseNode]]
    foreach n $parentNodes {
        set nodeMemberList [textnodes get node.$n.members]
        foreach p $nodeMemberList {
            set name [lindex [lindex $p 0] 0]
            if { [editor_SearchMemberListFor $memberList $name] == -1 } {
              lappend memberList [concat [list [list $name inherit]]     \
                 [lrange $p 1 end]]
            }
        }
    }
    return $memberList
}

# ---------------------------------------------------------------------
# The environment has changed in the editor
# ---------------------------------------------------------------------

proc editor_EnvironmentChanged key {
    global gNodeBinding

    set parts [split $key .]
    set keyItem [lindex $parts 0]
    set lastPart [lindex $parts end]

    cw_Debug "editor env change: $keyItem  $lastPart" cyan

    if { $keyItem == "node" } {
        set item [textnodes get $key]
        set node [editor_NodeToWidget $item]

        if { $lastPart == "parent" } {
            set thisWidget [editor_NodeToWidget [lindex $parts 1]]
            if { [$thisWidget cget -parent] == "" } {
                $thisWidget configure -parent $node
            } elseif { $item == "" } {
                $thisWidget breaklink
            }
```

```
      } elseif { $lastPart == "coupling" } {
         set thisWidget [editor_NodeToWidget [lindex $parts 1]]
         $thisWidget configure -coupling $item
      } elseif { $lastPart == "text" } {
         set thisWidget [editor_NodeToWidget [lindex $parts 1]]
         $thisWidget configure -icontext $item
      } elseif { $lastPart == "id" } {
         #
         # create a new node on the screen
         set x [textnodes get node.$item.x]
         set y [textnodes get node.$item.y]
         set parent [textnodes get node.$item.parent]
         nodeIcon $node -coupling none -widget [list newWidget          \
            grouptable] -envname $item -linkcolor blue -line arrow      \
            -canvas ".c" -coupling [textnodes get node.$item.coupling]
         $node configure -icontext [editor_GetNodeTextName $item normal]
         $node.icon configure -width 1.5c -height 1.5c -scrollid none
         set id [.c create window $x $y -window $node]
         [$node getenvironment] set icontype team
         groupIcon_DragIcon .c $node $x $y absolute
         $node setcanvasinfo $id $x $y
         #
         # see if anybody is already open in this node
         set currentPeople [textnodes get node.$item.currentPeople]
         foreach i $currentPeople {
            $node.icon adduser [lindex $i 0] [lindex $i 1]
         }
         $node bindall <Double-Button-1>                                \
            "editor_EnterNode .$item $item $node"
         $node bindall <1> "editor_StartDragIconImage $node"
         $node bindall <B1-ButtonRelease>                               \
            "editor_StopDragIconImage $node %X %Y"
         $node bindall <B1-Motion> "editor_DragIconImage $node %X %Y"
         $node bindall <Double-Button-2> "editor_NodeAttributes $node"
         $node bindall <B3-Motion>                                      \
            "editor_DragIcon .c $node %x %y relative"
         if { $gNodeBinding == 0 } {
            breakLink bind nodeIconBreakLink "editor_BreakLink %N %P"
            breakLink bind nodeIconCoordsChanged "editor_NewXY %N %X %Y"
            set gNodeBinding 1
         }
      }
   }
}

# ------------------------------------------------------------------------
# Drag an icon around the canvas
# ------------------------------------------------------------------------

proc editor_DragIcon { rootCanvas w x y state } {
   set node [editor_WidgetToNode $w]
   set ability [editor_GetAbility $node]
   if { $ability != [list no no no no] } {
      groupIcon_DragIcon $rootCanvas $w $x $y $state
   }
}

# ------------------------------------------------------------------------
# Node selection routine
# ------------------------------------------------------------------------
```

```
proc _editor_GetSelectedNodes { parent } {
   set ourList [editor_WidgetToNode $parent]
   set children [$parent cget -children]
   foreach i $children {
      lappend ourList [_editor_GetSelectedNodes $i]
   }
   return $ourList
}

proc editor_GetSelectedNodes {} {
   global gSelection

   if { $gSelection != "" } {
      return [_editor_GetSelectedNodes $gSelection]
   }
   return {}
}

# ---------------------------------------------------------------------
# Node coupling for tight/loose collaboration
# ---------------------------------------------------------------------

proc editor_Couple { value } {
   set nodes [editor_GetSelectedNodes]
   foreach node $nodes {
      textnodes set node.$node.coupling $value
   }
}

# ---------------------------------------------------------------------
# Create a new node. We use loose coupling initially by default.
# ---------------------------------------------------------------------

proc editor_NewNode { w x y } {

   if ![editor_CanAuthorAtAll] { return }

   set counter [cw_GetUniqueNum]
   set value "v$counter"

   texttree addbelow "" $value
   textnodes begintransaction node.$value.id
   textnodes set node.$value.id $value
   textnodes set node.$value.x [.c canvasx $x]
   textnodes set node.$value.y [.c canvasy $y]
   textnodes set node.$value.parent ""
   textnodes set node.$value.coupling none
   textnodes set node.$value.currentPeople {}
   textnodes set node.$value.sectionlevel ""
   textnodes set node.$value.autonumber 1
   textnodes set node.$value.valueset 0
   textnodes set node.$value.members                                    \
      [list [list [list [userprefs name] local] yes yes yes yes]]
   textnodes sendtransaction
   textnodes set node.$value.text [editor_GetNodeTextName $value force]
}

# ---------------------------------------------------------------------
# Draw the user interface
# ---------------------------------------------------------------------

proc editor_BuildDisplay {} {                                           \
   if { ![cw_CreateNamedImage [image types]
      /phd/gk/cw/bitmaps/peopleicon peopleicon] } {
      error "No peopleicon icon"
```

```
    }
    attrpicturebar_Init

    set project [string range [file extension                                    \
       [users get local.projectNode]] 1 end]
    wm title . "Project \"$project\""

    gk_defaultMenu .mbar
    canvas .c -scrollregion [list 0 0 900 1800]                                  \
       -xscrollcommand { .s2 set } -yscrollcommand { .s1 set }
    scrollbar .s1 -relief sunk -command { .c yview }
    scrollbar .s2 -relief sunk -command { .c xview } -orient horizontal

    .mbar add selectmenu 1 -text Selection
    .mbar itemcommand 1 add command -label "Tight coupling"                      \
       -command "editor_Couple tight"
    .mbar itemcommand 1 add command -label "No coupling"                         \
       -command "editor_Couple none"

    pack .mbar -side top -fill x
    pack .s2 -side bottom -fill x
    pack .s1 -side right -fill y
    pack .c -fill both -expand yes

    bind .c <Double-Button-1>    "editor_NewNode .c %x %y"
    bind .c <1>                  "editor_ClearSelection"
}

# ---------------------------------------------------------------------------
# Initialise environments
# ---------------------------------------------------------------------------

cw_Debug "** Starting Text Editor **" green

gk_newenv -share -bind textnodes
gk_newenv -share -bind views
gk_newenv -share texttree

cw_InitialiseTransaction textnodes
cwtree_InitialiseTree texttree

textnodes bind envReceived   [list editor_UpdateEnvironment]
textnodes bind addEnvInfo    [list editor_EnvironmentChanged %K]
textnodes bind changeEnvInfo   [list editor_EnvironmentChanged %K]

# ---------------------------------------------------------------------------
# Setup the colours for when a new user enters the world
# ---------------------------------------------------------------------------

gk_bind updateEntrant "editor_UpdateNewUser %U"

proc editor_UpdateNewUser newUsernum {
    global gProjectName

    cw_Debug "Updating user $newUsernum" orange
    gk_toUserNum $newUsernum set gProjectName $gProjectName
    gk_toUserNum $newUsernum SetCursorColourForOthers
}

# ---------------------------------------------------------------------------
# Build our user interface
# ---------------------------------------------------------------------------

editor_BuildDisplay
```

Appendix D
Collaborwriter negotiation tool

The code controls the main negotiation and consolidation environment, including issue creation and manipulation and shared drawing tools. The shared whiteboard is based on code contained within the GroupKit groupware toolkit.

D.1 Program code

```
#!/usr/local/bin/gkwish -f
# ------------------------------------------------------------------------
# negotiator.tool
# Negotiation tool for Collaborwriter
#
# Parts based on the shared whiteboard elements in GroupKit

# ------------------------------------------------------------------------
# The negotiator consists of a set of unrelated issues (in a list).
# For each issue (and its related issues), there is an ibisIcon node
# type where relationships get made between issues/positions/arguments.
# ------------------------------------------------------------------------

# ------------------------------------------------------------------------
# Initialise the conference process, linking it to the session manager
# ------------------------------------------------------------------------

source "[file dirname $argv0]/../library/envir.tcl"
source "[file dirname $argv0]/../library/conf.tcl"
gk_initConf $argv

source /usr/local/lib/groupkit/contrib/drawobjs/model.tcl
source "[file dirname $argv0]/../library/view.tcl"
source /usr/local/lib/groupkit/contrib/drawobjs/obj_line.tcl
source /usr/local/lib/groupkit/contrib/drawobjs/obj_rect.tcl
source /usr/local/lib/groupkit/contrib/drawobjs/obj_oval.tcl
source "[file dirname $argv0]/../library/freehand_line.tcl"
source "[file dirname $argv0]/../library/obj_text.tcl"

# ------------------------------------------------------------------------
# Set up autoloading to use appropriate library routines plus
# overriding routines in this file
# ------------------------------------------------------------------------

set rootDir [file dirname $argv0]/../library
set auto_path [linsert $auto_path 0 "/usr/local/lib/groupkit"]
set auto_path [linsert $auto_path 0 $rootDir]

unset rootDir

# foreach dir $auto_path { source "$dir/tclIndex" }
```

```
users local.persistenceStyle always

# ------------------------------------------------------------------------
# Globals for the current selection/hilite (pointless as environment?)
# ------------------------------------------------------------------------

set gLinkStyle text
set gNodeBinding 0
set gDraggingIcon 0
set gSelection ""
set gCurrentHilite ""
set gDefaultAbility UNDEFINED
set gProjectName ""
set gOrigTextID ""
set gInPatternNote 1
set gCurrentDrawingColour [userprefs color]

# ------------------------------------------------------------------------
# Fire off an event to get our initial abilities from the session mgr
# (ie. what we can do)
# ------------------------------------------------------------------------

if [gk_amOriginator] {
   keylset event type editorRequestsInitialInfo                          \
      confnum [users local.confnum] id [users local.originator]
} else {
   keylset event type editorRequestsInitialInfo                          \
      confnum [users local.confnum] id [users local.confnum]
}

cw_toRC $event

gk_on { ([gk_event type]=="sessmgrSendInfo") &&                          \
                        ([gk_event confnum]==[users local.confnum])} {
   if { [gk_event details] != "none" } {
      set gDefaultAbility [gk_event details]
   }

   textnodes set peopleInWorld [gk_event world]
   textnodes set peopleIcons [gk_event worldicons]
   textnodes set peopleColours [gk_event worldcolours]

   set gProjectName [gk_event proj]
   cw_Debug "Default ability: $gDefaultAbility" yellow
   cw_Debug "world: [gk_event world]" yellow
   cw_Debug "icons: [gk_event worldicons]" yellow
   cw_Debug "colours: [gk_event worldcolours]" yellow

   set myIdx [lsearch [gk_event world] [users local.username]]
   users set local.iconpict [lindex [gk_event worldicons] $myIdx]

   set myColour [lindex [gk_event worldcolours] $myIdx]

   SetCursorColourFor [users local.usernum] $myColour
}

# ------------------------------------------------------------------------
# Change our preference colour in all other processes
# ------------------------------------------------------------------------

proc SetCursorColourFor { usernum colour } {
   gk_setUserAttrib $usernum cursor_color $colour
}

proc SetCursorColourForOthers {} {
   gk_toOthers gk_setUserAttrib [users local.usernum]                    \
```

```
        cursor_color [users local.cursor_color]
}

# ------------------------------------------------------------------------
# Routine required by multitext to return the colour of user <name>
# ------------------------------------------------------------------------

proc GetColourForUserName { name } {
   set name [cw_AddSpacesToName $name]
   set peopleInWorld [textnodes get peopleInWorld]
   set idx [lsearch $peopleInWorld $name]

   if { $idx == -1 } {
      cw_Debug "--- Couldn't find user $name to get colour ---" red
      return black
   }

   return [lindex [textnodes get peopleColours] $idx]
}

# ------------------------------------------------------------------------
# Utility routine to get a boolean from a yes/no/inherit
# ------------------------------------------------------------------------

proc IsYes { thing } {
   if { $thing == "yes" } { return 1 }
   return 0
}

# ------------------------------------------------------------------------
# Utility routines for converting widget to node id and vice-versa
# ------------------------------------------------------------------------

proc negotiator_WidgetToNode { w } {
   set w [string range $w 3 end]
   set pos [string first "." $w]
   if { $pos != -1 } { set w [string range $w 0 [expr $pos - 1]] }
   return $w
}

proc negotiator_NodeToWidget { w } { return ".c.$w" }

# ------------------------------------------------------------------------
# link creation/deletion routines
# ------------------------------------------------------------------------

proc negotiator_NewXY { nodeW x y } {
   set node [negotiator_WidgetToNode $nodeW]
   textnodes set node.$node.x $x
   textnodes set node.$node.y $y
}

proc negotiator_BreakLink { nodeW parentW } {
   set parent [negotiator_WidgetToNode $parentW]
   set child [negotiator_WidgetToNode $nodeW]

   #
   # we can only break a link between nodes if we have author or
   # negotiator access to BOTH

   if { [negotiator_CanManipulate $parent] &&                    \
        [negotiator_CanManipulate $child] } {
      textnodes set node.$child.parent ""
   }
}
```

```
# ------------------------------------------------------------------------
# We assume the person can manipulate the node
# ------------------------------------------------------------------------

proc negotiator_CanManipulate { node } {
   return 1
}

# ------------------------------------------------------------------------
# Start dragging a node
# ------------------------------------------------------------------------

proc negotiator_StartDragIconImage { w } {
   global gSelection gDraggingIcon gCurrentHilite

   if { $gCurrentHilite != "" } { $gCurrentHilite highlight reset all }
   negotiator_ClearSelection
   set gSelection $w
   set gDraggingIcon 0
   set gCurrentHilite ""

   $w highlight yellow no
}

# ------------------------------------------------------------------------
# Clear any selection thats current
# ------------------------------------------------------------------------

proc negotiator_ClearSelection {} {
   global gSelection

   if { $gSelection != "" } {
      $gSelection highlight reset all
      set gSelection ""
   }
}

# ------------------------------------------------------------------------
# Stop dragging the image
# ------------------------------------------------------------------------

proc negotiator_StopDragIconImage { startW x y } {
   global gDraggingIcon gCurrentHilite

   set endW [negotiator_GetWidgetUnderMouse $startW $x $y]

   #
   # if we've been dragging the icon, lose the selection we created

   if { $gDraggingIcon == 1 } {
      . configure -cursor {}
      negotiator_ClearSelection
   }

   if { $gCurrentHilite != "" } {
      $gCurrentHilite highlight reset all
      set gCurrentHilite ""
   }

   if { $endW != "" } {
      set linkto [negotiator_WidgetToNode $endW]
      set linkfrom [negotiator_WidgetToNode $startW]

      #
      # to create the link we create a default link and then popup the
      # Change Link... dialog
```

```
        cw_NotImplementedYet "ibisicon drag & drop link"

        #
        # we assume we have negotiator access to the nodes
        # textnodes set node.$child.parent [textnodes get node.$parent.id]

    }
}

# ------------------------------------------------------------------------
# Actually drag the node pointer and see what we can drop onto
# ------------------------------------------------------------------------

proc negotiator_DragIconImage { startW x y } {
    global gDraggingIcon gCurrentHilite

    if { $gDraggingIcon == 0 } {
        set gDraggingIcon 1
        . configure -cursor [list @bitmaps/cursornode black]
    }

    set w [negotiator_GetWidgetUnderMouse $startW $x $y]
    if { $w != $gCurrentHilite } {
        if { $gCurrentHilite != "" } {$gCurrentHilite highlight reset all}
        set gCurrentHilite $w
        if { $gCurrentHilite != "" } { $gCurrentHilite highlight cyan no }
    }
}

# ------------------------------------------------------------------------
# See if we can create a link
# ------------------------------------------------------------------------

proc negotiator_GetWidgetUnderMouse { startW x y } {
    set node [negotiator_WidgetToNode $startW]

    set w [winfo containing $x $y]
    if { [string first ".c." $w] != 0 } { return "" }

    set w [string range $w 3 end]
    set pos [string first "." $w]
    if { $pos != -1 } { set w [string range $w 0 [expr $pos - 1]] }
    set w ".c.$w"

    if { [$w canattach $startW] == 0 } { return "" }
    return $w
}

# ------------------------------------------------------------------------
# Update the screen display rebuilding nodes as necessary
# ------------------------------------------------------------------------

proc negotiator_UpdateEnvironment {} {

    #
    # update the node data structures

    foreach node [textnodes keys node] {
        textnodes set node.$node.id [textnodes get node.$node.id]
    }
}

# ------------------------------------------------------------------------
# The environment has changed in the negotiator
# ------------------------------------------------------------------------
```

```
proc negotiator_EnvironmentChanged { key } {
   global gNodeBinding gLinkStyle gInPatternNote

   set parts [split $key .]
   set keyItem [lindex $parts 0]
   set lastPart [lindex $parts end]

   cw_Debug "negotiator env change: $keyItem  $lastPart" cyan

   if { $keyItem == "node" } {
      set item [textnodes get $key]

      if { $lastPart == "attachedNodes" } {
         set thisWidget [negotiator_NodeToWidget [lindex $parts 1]]
         [lindex $item 0] copycustom $thisWidget
         $thisWidget changeattachments $item
      } elseif { $lastPart == "coupling" } {
         set thisWidget [negotiator_NodeToWidget [lindex $parts 1]]
         $thisWidget configure -coupling $item
      } elseif { $lastPart == "text" } {
         set thisWidget [negotiator_NodeToWidget [lindex $parts 1]]
         # $thisWidget configure -icontext $item
      } elseif { $lastPart == "id" } {

         #
         # create a new node on the screen

         set node [negotiator_NodeToWidget $item]
         set x [textnodes get node.$item.x]
         set y [textnodes get node.$item.y]
         set entity [textnodes get node.$item.entity]

         ibisIcon $node -coupling none -widget [list newWidget label]  \
            -envname $item -canvas ".c" -entity $entity                \
            -linkstyle $gLinkStyle -scrollid $item                     \
            -coupling [textnodes get node.$item.coupling]

         if { $gInPatternNote } {
            $node configure -strict no -showvotes no
         }

         $node configure -menucmd negotiator_NodeMenu -text "$entity"
         $node.icon configure -wraplength 120
#         $node.icon insert 1.0 "This is a $entity"
#         $node.icon configure -coupling none -scrollid $item          \
#            -envname $item -coupling tight -width 10 -height 2

         set id [.c create window $x $y -window $node]
         [$node getenvironment] set icontype team
         groupIcon_DragIcon .c $node $x $y absolute
         $node setcanvasinfo $id $x $y

         #
         # see if anybody is already open in this node

         set currentPeople [textnodes get node.$item.currentPeople]
         foreach i $currentPeople {
            $node.icon adduser [lindex $i 0] [lindex $i 1]
         }

         $node bindall <1> "negotiator_StartDragIconImage $node"
         $node bindall <B1-ButtonRelease>                              \
            "negotiator_StopDragIconImage $node %X %Y"
         $node bindall <B1-Motion>                                     \
            "negotiator_DragIconImage $node %X %Y"
```

```
            $node bindall <B3-Motion>                                    \
               "negotiator_DragIcon .c $node %x %y relative"

            if { $gNodeBinding == 0 } {
               breakLink bind nodeIconBreakLink                          \
                  "negotiator_BreakLink %N %P"
               breakLink bind nodeIconCoordsChanged                      \
                  "negotiator_NewXY %N %X %Y"
               set gNodeBinding 1
            }
         }
      }
   }

# ---------------------------------------------------------------------
# Move the node around the canvas
# ---------------------------------------------------------------------

proc negotiator_DragIcon { rootCanvas w x y state } {
   set node [negotiator_WidgetToNode $w]
   groupIcon_DragIcon $rootCanvas $w $x $y $state
}

# ---------------------------------------------------------------------
# Find out what nodes are currently selected
# ---------------------------------------------------------------------

proc _negotiator_GetSelectedNodes { parent } {
   set ourList [negotiator_WidgetToNode $parent]
   set children [$parent getattachments]
   foreach i $children {
      lappend ourList [negotiator_WidgetToNode [lindex $i 0]]
   }
   return $ourList
}

proc negotiator_GetSelectedNodes {} {
   global gSelection

   if { $gSelection != "" } {
      return [_negotiator_GetSelectedNodes $gSelection]
   }
   return {}
}

# ---------------------------------------------------------------------
# Menu option for viewing ibisicon links as text or icons
# ---------------------------------------------------------------------

proc negotiator_ViewLinksAs {} {
   global gLinkStyle

   set nodes [negotiator_GetSelectedNodes]
   if { $nodes == "" } { set nodes [textnodes keys node] }

   foreach i $nodes {
      set widget [negotiator_NodeToWidget $i]
      $widget configure -linkstyle $gLinkStyle
   }
}

# ---------------------------------------------------------------------
# Update the new users data (and ours) when he arrives
# ---------------------------------------------------------------------
```

```
proc negotiator_UpdateNewUser newUsernum {
   global gProjectName

   cw_Debug "Updating user $newUsernum" orange
   gk_toUserNum $newUsernum set gProjectName $gProjectName
   gk_toUserNum $newUsernum SetCursorColourForOthers
}

# ---------------------------------------------------------------------
# Create a new node. We use tight coupling by default.
# ---------------------------------------------------------------------

proc negotiator_NewNode { x y entity link linkedTo coordtype } {

   set counter [cw_GetUniqueNum]
   set value "v$counter"

   if { $coordtype == "screen" } {
      set x [.c canvasx $x]
      set y [.c canvasy $y]
   }

   textnodes begintransaction node.$value.id
   textnodes set node.$value.id $value
   textnodes set node.$value.x $x
   textnodes set node.$value.y $y
   textnodes set node.$value.entity $entity
   textnodes set node.$value.link $link
   textnodes set node.$value.attachedNodes {}

   textnodes set node.$value.coupling tight
   textnodes set node.$value.currentPeople {}
   textnodes set node.$value.sectionlevel ""
   textnodes set node.$value.autonumber 1
   textnodes set node.$value.valueset 0
   textnodes set node.$value.members                                    \
      [list [list [list [userprefs name] local] yes yes yes yes]]
   textnodes sendtransaction
   textnodes set node.$value.text $entity

   if { $linkedTo != "nil" } {
      textnodes set node.$value.attachedNodes                           \
         [list [negotiator_NodeToWidget $linkedTo] $link]
   }
}

# ---------------------------------------------------------------------
# Callback routine for the ibisicon to create a new widget on result of
# a popup menu selection
# ---------------------------------------------------------------------

proc negotiator_NodeMenu { widget entity link } {
   set callingNode [negotiator_WidgetToNode $widget]
   set env [$widget getenvironment]
   set coords [$env get coords]
   set x [expr [lindex $coords 0] + 50]
   set y [expr [lindex $coords 1] + 50]

   negotiator_NewNode $x $y $entity $link $callingNode canvas
}

# ---------------------------------------------------------------------
# Show/hide the drawing tools at the bottom of the window
# ---------------------------------------------------------------------
```

```
proc negotiator_ToggleDrawingTools {} {
   global gToolbar gInPatternNote

   if { $gToolbar == 1 } {
      pack .toolbar -before .s2 -side bottom -fill x -expand no
      foreach i [winfo children .toolbar] {
         pack $i -side left
      }

      if { $gInPatternNote == 0 } {
         pack forget .toolbar.mbutton
      }
   } else {
      pack forget .toolbar
   }
}

# ---------------------------------------------------------------------
# In the whiteboard, rebuild the original text view
# ---------------------------------------------------------------------

proc negotiator_UpdateOriginalText {} {
   global gOrigTextID gCurrentDrawingObject

   if { $gOrigTextID == "" } {

      #
      # there's been no creation of the original text, so create the
      # object

      view newobjtype origtext
      set gOrigTextID [view _startsweep 50 100]

   } else {
      cw_NotImplementedYet "Update text in whiteboard - $gOrigTextID"
   }

   update idletasks
   view newobjecttype none
   set gCurrentDrawingObject select
}

# ---------------------------------------------------------------------
# Change the current menu colour for drawing if we're in pattern note
# mode
# ---------------------------------------------------------------------

proc negotiator_SetDrawingColour { colour } {
   global gCurrentDrawingColour
   .toolbar.mbutton configure -bg $colour
   set gCurrentDrawingColour $colour
}

# ---------------------------------------------------------------------
# Tell the whiteboard view what colour it should use
# ---------------------------------------------------------------------

proc negotiator_WhatColourToUse {} {
   global gInPatternNote gCurrentDrawingColour
      if { $gInPatternNote == 1 } {
      return $gCurrentDrawingColour
   } else {
      return [userprefs color]
   }
}
```

```
# ------------------------------------------------------------------------
# Build our negotiator window display
# ------------------------------------------------------------------------

proc negotiator_BuildDisplay {} {
   global gLinkStyle gToolbar gCurrentDrawingObject gInPatternNote

   set project [string range                                              \
      [file extension [users get local.projectNode]] 1 end]
   wm title . "Negotiator for Project \"$project\""

   gk_defaultMenu .mbar
   listbox .l -yscrollcommand { .s set } -background white -width 10    \
      -exportselection 0 -selectbackground blue -selectforeground yellow
   scrollbar .s -relief sunk -command { .l yview }
   entry .m -background yellow -foreground red
   canvas .c -scrollregion [list 0 0 900 1800]                           \
      -xscrollcommand { .s2 set } -yscrollcommand { .s1 set }
   scrollbar .s1 -relief sunk -command { .c yview }
   scrollbar .s2 -relief sunk -command { .c xview } -orient horizontal

   .mbar add issuemenu 1 -text Issues
   .mbar itemcommand 1 add command -label "No issues..."
   .mbar itemcommand 1 add checkbutton -label "Pattern note"             \
      -variable gInPatternNote -command "negotiator_ToggleDrawingTools"

   .mbar add viewmenu 2 -text "View"
   .mbar itemcommand 2 add radiobutton -label "Links as text"            \
      -variable gLinkStyle -value text   -indicatoron 0                  \
      -command [list negotiator_ViewLinksAs]
   .mbar itemcommand 2 add radiobutton -label "Links as icons"           \
      -variable gLinkStyle -value icon   -indicatoron 0                  \
      -command [list negotiator_ViewLinksAs]
   .mbar itemcommand 2 add checkbutton -label "Drawing toolbar"          \
      -variable gToolbar -command [list negotiator_ToggleDrawingTools]

   bind .c <Double-Button-1>                                             \
               "negotiator_NewNode %x %y issue none nil screen"
   bind .c <1>   "negotiator_ClearSelection"

   #
   # create the toolbar and its buttons

   set b "/phd/gk/cw/bitmaps"
   frame .toolbar
   radiobutton .toolbar.select  -bitmap "@$b/draw_select"                \
      -variable gCurrentDrawingObject -indicatoron 0 -value select       \
      -selectcolor yellow -command "view newobjtype none"
   radiobutton .toolbar.freehand -bitmap "@$b/draw_freehand"             \
      -variable gCurrentDrawingObject -indicatoron 0 -value freehand     \
      -selectcolor yellow -command "view newobjtype freehand"
   radiobutton .toolbar.line -bitmap "@$b/draw_line"                     \
      -variable gCurrentDrawingObject -indicatoron 0 -value line         \
      -selectcolor yellow -command "view newobjtype line"
   radiobutton .toolbar.oval -bitmap "@$b/draw_oval" -value oval         \
      -variable gCurrentDrawingObject -indicatoron 0                     \
      -selectcolor yellow -command "view newobjtype oval"
   radiobutton .toolbar.rect -bitmap "@$b/draw_rect"                     \
      -variable gCurrentDrawingObject -indicatoron 0 -value rectangle    \
      -selectcolor yellow -command "view newobjtype rectangle"
   radiobutton .toolbar.text -bitmap "@$b/draw_text"                     \
      -variable gCurrentDrawingObject -indicatoron 0 -value text         \
      -selectcolor yellow -command "view newobjtype text"
   entry .toolbar.entry -textvariable gText
```

```
    radiobutton .toolbar.origtext -bitmap "@$b/draw_origtext"          \
       -variable gCurrentDrawingObject -indicatoron 0 -value origtext  \
       -selectcolor yellow -command "negotiator_UpdateOriginalText"
    menubutton .toolbar.mbutton -text "Colour" -indicatoron 0          \
       -menu .toolbar.mbutton.m
    menu .toolbar.mbutton.m

    set c [userprefs color]
    .toolbar.mbutton.m add command -label "Default" -background $c     \
       -command [list negotiator_SetDrawingColour $c]
    .toolbar.mbutton.m add separator

    foreach i { lavender SlateGrey MidnightBlue NavyBlue blue LightBlue
                cyan aquamarine SeaGreen green khaki yellow goldenrod
                orange IndianRed peru red salmon tomato DeepPink maroon
                magenta DarkOrchid thistle black white } {
       .toolbar.mbutton.m add command -background $i                   \
          -command [list negotiator_SetDrawingColour $i]
    }

    ## create the model, view and controller for the whiteboard

    gk_objmodel model
    gk_objview view .c model

    ## add item types we want

    view addtype freehand freehandline
    view addtype line stdline
    view addtype rectangle stdrect
    view addtype oval stdoval
    view addtype text stdtext
    view addtype origtext stdtext

    set gCurrentDrawingObject select
    view newobjtype none

    pack .mbar -side top -fill x
    pack .l .s -side left -fill y

    negotiator_ToggleDrawingTools

    pack .m -side top -fill x
    pack .s2 -side bottom -fill x
    pack .s1 -side right -fill y
    pack .c -fill both -expand yes
}

# ---------------------------------------------------------------------
# The starting and initialisation routines...
# ---------------------------------------------------------------------

cw_Debug "** Starting Negotiator **" green

gk_newenv -share -bind textnodes
gk_newenv -share -bind views

cw_InitialiseTransaction textnodes

textnodes bind envReceived   [list negotiator_UpdateEnvironment]
textnodes bind addEnvInfo   [list negotiator_EnvironmentChanged %K]
textnodes bind changeEnvInfo   [list negotiator_EnvironmentChanged %K]

gk_bind updateEntrant "negotiator_UpdateNewUser %U"

negotiator_BuildDisplay
```

Appendix E

Collaborwriter groupware widgets

The widget code is divided into separate code modules for each widget or widget group.

E.1 GroupIcon generic groupware widget

```
# ------------------------------------------------------------------------
# groupicon.tcl
#
# Base groupware widget with underlying environment data structure

# ------------------------------------------------------------------------
# groupIcon
# This is a groupware widget consisting of an icon with some text
# underneath it. It can be dragged around its root canvas with the
# righthand mouse button.
# It can be loosely or tightly coupled, and uses an environment to
# decide what to do.
#
# It's option fields are:
#   envname     - name of the underlying environment
#   widget      - icon to display or [list newWidget <widget type>]
#   icontext    - label under the icon
#   coupling    - none | tight
#   canvas      - the canvas the icon is on (this sets up the drag
#                 binding as well)
#   defaulticon - a default image to use (**MUST BE VALID**)
#
# It's methods are:
#   getenvironment - return the underlying environment data structure
#   setcanvasinfo  - set up its canvasID and x,y coords (MUST BE CALLED)
#   bindall        - apply bindings on all the icons in the widget
#   setstate       - set a cfg option in all widgets (e.g. -bg colour)
#   getstate       - return the settings of an option for all nodes
#                    (eg [w red][w.x blue])
# ------------------------------------------------------------------------

# ------------------------------------------------------------------------
# Initialise and construct the widget
# ------------------------------------------------------------------------

proc groupIcon { w args } {
   eval gkInt_CreateWidget $w groupIcon groupIconClass $args
   return $w
}

proc groupIcon_CreateClassRec {} {
   global groupIcon

   set groupIcon(inherit)  {label}
   set groupIcon(options)  {-envname -widget -icontext -coupling -canvas
```

```
                                -defaulticon}
    set groupIcon(methods)  {getenvironment setcanvasinfo bindall
                             setstate getstate}

    set groupIcon(-envname)     {-envname envName EnvName ""}
    set groupIcon(-widget)      {-widget widget Widget error}
    set groupIcon(-icontext)    {-icontext iconText IconText <icon>}
    set groupIcon(-coupling)    {-coupling coupling Coupling forcenone}
    set groupIcon(-canvas)      {-canvas canvas Canvas .}
    set groupIcon(-defaulticon) {-defaulticon defaultIcon DefaultIcon ""}
}

proc groupIcon_InitWidgetRec { w class className args } {
    upvar #0 $w groupIcon
    set groupIcon(environment) ""
}

proc groupIcon_ConstructWidget { w } {
    upvar #0 $w groupIcon

    $w configure -relief raised -background grey -borderwidth 2
    label $w.icon -image                                                    \
        [lindex $groupIcon(-widget) 3] -background grey -takefocus 1
    label $w.icontext -text [lindex $groupIcon(-icontext) 3]                \
       -background grey -takefocus 1 -justify center -wraplength 75         \
       -font "-*-lucida-medium-r-normal-*-10-*-*-*-*-*-*-*"
    pack $w.icon $w.icontext
}

proc groupIcon_DestroyWidget { w } {
    upvar #0 $w data
    if { $data(environment) != "" } { $data(environment) destroy }
}

# ------------------------------------------------------------------------
# Configure the widget
# ------------------------------------------------------------------------

proc groupIcon_Config { w option args } {
    upvar #0 $w groupIcon
    set args [lindex $args 0]

    switch -exact [string range $option 1 end] {
       envname  {

          #
          # Get the internal environment & bindings done now

          if { $args != "" } {
             set env1 "gIcn_$args"
             regsub -all " " $env1 "_" env
             groupIcon_BuildEnvironment $w $env
          }
       }

       icontext { $w.icontext configure -text $args }

       widget {

          #
          # Change the icon picture to a different image. If the image
          # doesn't exist, first try to load it in (the image name is the
          # filename of the image. Otherwise use the default icon defined
          # using -defaulticon

          # now we can override the widget and replace it with
          # something else
```

```
            if { [lindex $args 0] == "newWidget" } {
               destroy $w.icon
               [lindex $args 1] $w.icon -takefocus 1
               pack $w.icon -before $w.icontext
            } else {
               set currentImgs [image names]
               set img $args
               if { [lsearch $currentImgs $img] == -1 } {
                  if { [cw_CreateImage [image types] $img] == 0 } {
                     set img $groupIcon(-defaulticon)
                  }
               }
               $w.icon configure -image $img
            }
         }

         coupling {
            if { ($args == "forcenone") } { return
            } elseif { $groupIcon(environment) == "" } {
               cw_Debug "coupling call: $args with no env built" red
            } elseif { $args == "none" } {
               cw_UncoupleEnvironment $groupIcon(environment)
            } else {
               cw_CoupleEnvironment $groupIcon(environment)
            }
         }

         canvas   {
            $w bindall <B3-Motion>                                          \
               "groupIcon_DragIcon $args $w %x %y relative"
         }

         defaulticon {}

         default  { $groupIcon(rootCmd) configure $option $args }
      }
   }

   # ------------------------------------------------------------------------
   # Build the uderlying environment for the widget - its this that gives
   # us most of our coupling facilities
   # ------------------------------------------------------------------------

   proc groupIcon_BuildEnvironment { w env } {
      upvar #0 $w groupIcon

      #
      # This is where we create our environment, as it uses the label name
      # to build it.
      # We also create our bindings for the widget here

      cw_Debug "Build $env, widget: $w, couple: $groupIcon(-coupling)" pink

      if { $groupIcon(-coupling) == "forcenone" } {

         # special case for registrar client icons
         # (where envs can't be shared)

         gk_newenv -bind $env
      } else {

         # normally we create the icon as coupled and then turn off the
         # coupling

         gk_newenv -bind -share $env
         cw_InitialiseCoupling $env
```