Supporting Intellectual Work through Rendering and Review

Lars Yde

11 April 2001

Contents

Chapter 1

Introduction and background

Software development is difficult. This observation comes not only from my personal experience, but seems to be universally acknowledged by practitioners and researchers in the field alike [11, 51, 19]. Nor is it a particularly novel or sensational insight: numerous manuscripts and much research have recognized it, addressed it and, in many cases, proffered some panacea for remedying it. The problem persists, though, with no signs of abating. Software systems continue to be late, continue to frustrate their intended users and continue to be flawed or even dysfunctional [21]. The most important reason for what seems to be a perennial problem is probably the inherent complexity of software development with its complicated interplay of technological, social and psychological factors. The difficulty in managing this complexity is severely compounded by the intangibility of software engineering's subject matter. Contrary to other branches of engineering which are typically concerned with the transformation of physical matter in one form or another, software engineering is concerned with the creation of symbolic artifacts. In other words, software engineers have no immediate sensory perception of the product they fashion because it is, in essence, a purely intellectual construct. In his much quoted 1987 article [11], Fred Brooks states it so:

> "The reality of software is not inherently embedded in space. Hence, it has no ready geometric representation in the way land has maps, silicon chips have diagrams, computers have connectivity schematics ... In spite of progress in restricting and simplifying the structures of software, they remain inherently unvisualizable, and thus do not permit the mind to use some of its most powerful conceptual tools. This lack not only impedes the process of design within one mind, it severely hinders communication among minds."

Instead of direct observation, we rely on tools (compilers, debuggers, code analyzers, benchmarks, etc.) to help us gauge and probe the end product of a software development process, namely the code itself[1]. Such external measurement and experimentation can only process the syntactic and (formal) semantic properties of code. It does not yield information about its "sanity", that is, whether it is sensible relative to its problem domain or whether it makes

[1] A similar argument can be made for other phases, such as design and analysis but I will focus on program code for now.

logical sense[2]. To that end, a developer must still rely on human review conducted by herself or by others, either informally or systematized as code inspection meetings, walkthroughs or the like [40]. The value of review and scrutiny of program code and other symbolic artifacts, particularly by others than the author herself, should not be underestimated. Mathematicians have known and applied this knowledge for centuries through the institutionalized practice of proving the logical consistency of their ideas and then subjecting those proofs to scrutiny by other mathematicians. Software engineering has rediscovered the value of this mechanism and has accumulated statistical evidence to support it. For example, in [30] Humphrey writes that

> " ...if you were to inspect a software product that contained 100 defects, how many would you expect to find? We will call the percentage of the defects found the yield of the review or inspection. Again, there are no published data but my experience at IBM was that inspections typically yielded between 60 percent to 80 percent. Data from another organization support this with a reported 68% yield for one large operating system."

These characteristics of being intangible and subject to improvement through review is of course not exclusive to software. All artifacts of intellectual work seem to share them, consider for example coauthoring on scientific articles, proof-reading by editors, paper submissions by students to teachers, and so on. Software development is, however, conducted on an industrial scale and produces systems whose complexity dwarfs that of most other human artifacts. It might therefore be the area that could benefit the most from technologies, for example in the form of software tools, that could make software artifacts perceptible to its developers and thus more susceptible to both traditional and entirely new forms of inspection and review as the artifacts were being developed. Ideally, such tools would render a representation of a set of artifacts and the process by which they are brought about and would provide facilities for the generation and accumulation of review material concerning that representation. That is, in its fully fledged incarnation, the support tool envisioned would provide a both spatial and temporal rendition of a software project and facilities for communicating about it.

In Chapters 2 and 3 of this thesis, I present and discuss a prototype for such a tool, called PeerView, which was implemented by me in the latter half of 2000. The first part of that presentation will be an overview of this prototype in the form of an article coauthored with Jyrki Katajainen, which appeared as [62] and has been submitted to an international journal[3]. It is followed by a more thorough discussion of the material it introduces. In Chapter 5, I analyze the design, implementation and development process of the PeerView project, and discuss what lessons can be derived from it. Next, in Chapter 6, I apply those lessons to the design of a more ambitious system that is to succeed and in part build on the PeerView prototype. This system, which has been given the working title *InSiter*, will be implemented if the necessary resources can be made available, and will hopefully then be subjected to extensive usability testing. The thesis concludes with a brief summary and look at results achieved and experience garnered as well as a perspective on the future of this work.

[2]The "sanity" of a design, a piece of code or other symbolic information might be called its *natural semantics* to distinguish it from formalized descriptions of its semantics.

[3]Another, somewhat shorter article appeared in the February issue of Distributed Systems Online [61]

1.1 Acknowledgements

Thanks to my supervisor Jyrki Katajainen who provided guidance and ample room for discussion, both technical and otherwise, and who helped ensure that i's were dotted and t's crossed by proof-reading the draft manuscript. Also, through financial support from grant 9801749 (project Performance Engineering), he made it possible for me to focus on thesis writing, not making ends meet. Thanks also to Morten Nielsen who gave useful feedback on late drafts of the manuscript which helped improve the final version. The three participants in the experimental evaluation — Paul Broholm Abrahamsen, Thomas Hyldahl and Rikke Iversen — provided valuable oral and written feedback by allowing me to tape record our communication during the experiment and by subsequently completing a questionnaire. I would like to thank them also for their assistance. The Department of Computing at the University of Copenhagen (DIKU) proved helpful by providing me with office facilities and funds for external consultancy on portions of the thesis manuscript. Thanks also to Kasper Hornbæk for some interesting feedback on the PeerView interface. Finally, thanks to my girlfriend Jeanette Vedelstjerne Gersager who helped me regain what I was at times lacking in moral fibre.

Errors, omissions and typos are hopefully few and far between, but whatever may persist are of course entirely of my doing.

Chapter 2

Purpose, methodology and overview

2.1 Introduction

This Chapter provides an overview of the prototype software that was developed for this thesis and the problems that it addresses. This Chapter also includes a description of the work methodology used for the present thesis, i.e. what was the intended progression of events from inception to conclusion. The concluding Chapter (Chapter 7) assesses how this methodology was implemented, i.e. how was the correlation between what was intended and what actually happened.

2.2 The purpose of PeerView

PeerView is an exploratory prototype. Its purpose is to provide groups of users involved in intellectual work with a *common information space* and facilities for communicating about the contents of that information space. The notion of a common information space (CIS) is not unequivocally defined in existing literature, but in [4], Bannon and Bødker discuss the concept and give a description that can be summed up for the purposes of this work by the following quote from [48]:

> " ...a central archive of organizational information with some level of 'shared' agreement as to the meaning of this information (locally constructed), despite the marked differences concerning the origins and context of these information items."

They go on to note that:

> "Cooperative work is not facilitated simply by the provision of a shared database, but requires the active construction by the participants of a common information space where the meanings of the shared objects are debated and resolved, at least locally and temporarily."

PeerView is primarily intended as a CIS system for software developers since the artifacts of their shared work are often voluminous and it can be difficult to formulate the mental equivalent of the physical representation that manual workers have of their work simply by virtue of their artifacts being physical. In other words, if one's subject matter are strings of symbols, the very act of making sense of what one is working on takes effort, whereas with physical matter, the act of making sense is an unconscious "making of sense" through processing of sensory data that does not require the sometimes painstaking intellectual effort needed to understand complex symbolic representation. Common information spaces thus seem to derive their justification from the way in which they can help automate the work of constructing and maintaining representations of external activities, namely shared work. The potential benefits are not only the savings in work time and effort that one would suspect to see when individual collaborators can more quickly and systematically access and absorb the information needed to carry out their work, but also the organizational rewards that can potentially be reaped from an integrative model of shared work. By the latter I mean that when workers and management can access a comprehensive and coherent model of the work that goes on within their organization or group, they acquire an object of both conversation and contention. They can discuss and monitor progress, either informally or by means provided by the workspace system itself, and can review the shared artifacts themselves in a more immediate way than if they were not accessible at a central locus. It seems plausible that this can ultimately affect not only the work process itself, but also the organization within which that process is undertaken since such a model can give all stakeholders in the organization a common focal point for discussion. The above is largely speculative and the intention behind PeerView is not only to give users a support tool for their collaborative work, but also to explore how these ideas can best be translated into reality.

In [27], Gutwin and Greenberg refer to a particular type of common information space, namely the *shared worksspace* as "a bounded space where people can see and manipulate artifacts related to their activities." [27, Section 2.2]. They go on to note that "In these spaces, the focus of the activity is on the task artifacts: the visible and manipulable objects through the task is carried out." [27, Section 2.3]. This form of common information space is limited in that it usually seeks to reproduce or simulate a physical workspace such as a desktop or a conference room. Other forms of shared information, such as organizational structure or awareness of where coworkers are physically located and what they are doing, need not be present in the shared workspace or even deducible from it. If it is, the relationship between workspace artifacts and such awareness information might be entirely coincidental. PeerView's dominant feature is such a shared workspace where users can share and manipulate artifacts. The purpose of providing this workspace is both to give users a window onto the work of whatever group they are currently associated with and to experiment with the design and implementation of this type of common information space. The latter has yielded useful information, both through the process of development itself and through experimental evaluation, as discussed in Chapters 4 and 5, respectively. This information can be fed into the development of other systems that use common information spaces to support collaboration, one of which is outlined in Chapter 6.

In addition, PeerView demonstrates novel methods of navigation, at least in a groupware context, by giving users a fully zoomable desktop on which the shared artifacts can be placed and manipulated. This is an experimental way of addressing the "detail/overview" problem of how to represent both local detail and global overview in a usable manner which is discussed in [22], where the authors describe a different solution to this problem, namely the fisheye lens. PeerView also addresses the practical problem of catering to groups operating in a heterogeneous computing environment by being implemented in pure Java, i.e. without any native code, and therefore able to run on a variety of machine architectures and platforms.

Being a prototype, PeerView also helps answer the development question: how can a fully fledged CIS support system best be developed? PeerView may be limited in its functionality but it incorporates all the components that should be present in more ambitious systems requiring many man-years of development time, and the experience gained during all phases of PeerView development can facilitate implementation of such systems. Chapters 3 and 5 describe the development process and discuss what lessons can be learned from it.

PeerView imposes little structure on its users and their work process. This means that it affords users a great degree of freedom which is in keeping with an observation by Grintner [23]: "one of the emergent trends in workflow research ... [is] to make the systems more flexible to accomodate the contingent aspects of work.". However, it also means that PeerView does little to systematize and automate work for its users. Later systems, such as that described in Chapter 6, will have to address the specific needs of a more narrowly defined user audience to go beyond the simple functionality found in PeerView. In a sense, this implies a range of possible applications from the pure, unadorned support of shared information in a CIS to a highly structured, user customizable workflow system centered around a CIS. Viewed in that light, PeerView is clearly towards the former end of the spectrum. For that same reason, PeerView may appeal to a broad audience but does not purport to appeal to an equally broad range of work situations. That is, although PeerView may be used by anyone from the office clerk with basic computer literacy to the veteran software developer, it does not necessarily support all of their work situations and nor does it have to in order to fulfill its purpose.

2.3 Methodology

The methodology used in this project is a hybrid resulting from the need to reconcile scientific exploration with the practicalities of software development. The idea for a distributed application supporting intellectual work through rendering and review sprang from reading [52] among other works, and from personal experience with software development projects where lack of communication and coordination can complicate or derail efforts. Now, this conception of an idea could have been followed by lengthy elaboration process where the initial ideas were translated into a detailed paper model, i.e. a verbal and diagrammatic description, and not necessarily implemented as an executable application. This approach would obviously have allowed for a much more ambitious initial design since the lenghty implementation cycle of "code-execute-debug" could have been eliminated. However, I believed such an approach to be of questionable scientific value at best and worthless at worst since a pure "armchair design" that was not put to an experimental test through implementation would be much like a physics theory that was never empirically tested, i.e. insubstantial though possibly intellectually stimulating. Also, most software development methodologies advocate incremental or iterative development [6, 42, 32] which corresponds to my personal experience which is that most system development benefit from many short iterations rather than monolithic, phased design as in the traditional "waterfall" model [51, p. 9]. I therefore decided to advance quickly from idea to design and implementation, followed by *a posteriori* analysis of the development process and evaluation of the finished prototype in the hope that this would yield valuable feedback based on empirical data that could help improve the design and development of future systems.

Putting a methodology on a summary form as done below is slightly misguided as methodology is more a mindset and structured approach than a formulaic procedure. However, it is illustrative to see the overall steps taken put into a list form so that is done here:

1. Inception of PeerView
2. Elaboration of PeerView
3. Construction of PeerView
4. Transition
5. Evaluation
6. A posteriori analysis
7. Elaboration of successor system (next iteration)

To anyone familiar with the unified software development process [32], the above sequence will be recognizable, at least in part, because the activities “Inception”, “Elaboration”, “Construction” and “Transition” are identical to the phases in a “workflow cycle” to use unified process parlance. However, this is not in itself of scientific merit since these developmental phases do not include formal evaluation of the end product and the process by which it was brought about, and therefore does little to verify their validity. The evaluation and analysis phases of the process used for this work are intended to address this shortcoming and help give this thesis the scientific potency needed to have an impact beyond the software product resulting from it. In Chapter 7, I discuss how the actual progression of events corresponds with the process intended and what room there might be for improvement.

2.4 Supporting Intellectual Work Through Artifact Rendering and Group Review

The following is a slightly abbreviated version of an article that appeared as [62] and has been submitted to an international journal. Only the introduction has been cut to avoid overlap with Section 1. The article appears here to provide an overview of the PeerView software.

Supporting Intellectual Work through Artifact Rendering and Group Review

Lars Yde
Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
`larsyde@diku.dk`

Jyrki Katajainen
Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
`jyrki@diku.dk`

Abstract. Intellectual teamwork, such as that done by software development teams, is characterized by the intangibility of its subject matter. This can make it difficult for team members and outside stakeholders to gain an overview of the product being built. Many software tools of various designs have addressed this problem, often by using graphs and charts to model ongoing projects. In this paper, we suggest a design based on artifact rendering and group review and argue that it can promote overview and communication. We also present software built from that design and describe its potential uses and audience as well as its implementation.

Keywords.Computer-supported collaborative work, component-based development

Category.H.4.3, H.5.3, K.4.3

2.5 Overview

In [Section 2.6] we describe a prototype of a tool intended to support *intellectual teamwork* in general. That is, we aim at supporting any process in which a symbolic product, an *artifact*, is being produced by individuals in group collaboration, so the use is not limited to software developers. Our main design goal was to promote overview by artifact rendering and communication by group review. By *artifact rendering* we mean the representation of intellectual artifacts by some medium to facilitate processing by humans, and by *group review* the activity through which team members exchange information and commentary on the artifact being built. Our prototype uses visualization to render an artifact that consists of textual and graphical *documents*, but other forms can be envisaged, i.e. auditory representation. Also, discussion fora can be associated with parts of the artifact rendered.

Our prototype was developed using the Java programming language and a set of compatible component technologies. In [Section 2.7] we discuss both and argue that implementation was possible only through such component-based development, given our limited resources. Next, in [Section 2.8], we propose some potential uses of the prototype, both in its present form and in subsequent versions. We then go on to discuss earlier related work in [Section 2.9] and finally, in [Section 2.10], we outline our plans for a more advanced system.

2.6 Description of the Prototype

The prototype aimed at fulfilling the basic needs of artifact rendering and group review was named *PeerView* and work on it was begun in the middle of May 2000 as part of Lars Yde's M. Sc. thesis. At the time of writing (mid October), a beta release has been completed and a stable version is planned for release before the end of the year. The system is released under a freeware license both in its binary and source form. In this section, we outline PeerView's design.

2.6.1 System Overview

PeerView provides users with a dynamic representation of on-going work by rendering the artifact of that work in a form which allows easy overview as well as inspection and discussion of detail. PeerView accomplishes this by allowing groups of users to share documents and have them rendered in a scalable visual panorama. The set of documents is constantly kept updated so as to provide a real-time window onto the artifact (e.g., a collection of code files) being developed. With each document is associated a discussion forum that enables a group to have discussions on the document in a manner similar to a USENET [57] newsgroup having discussions of a set topic.

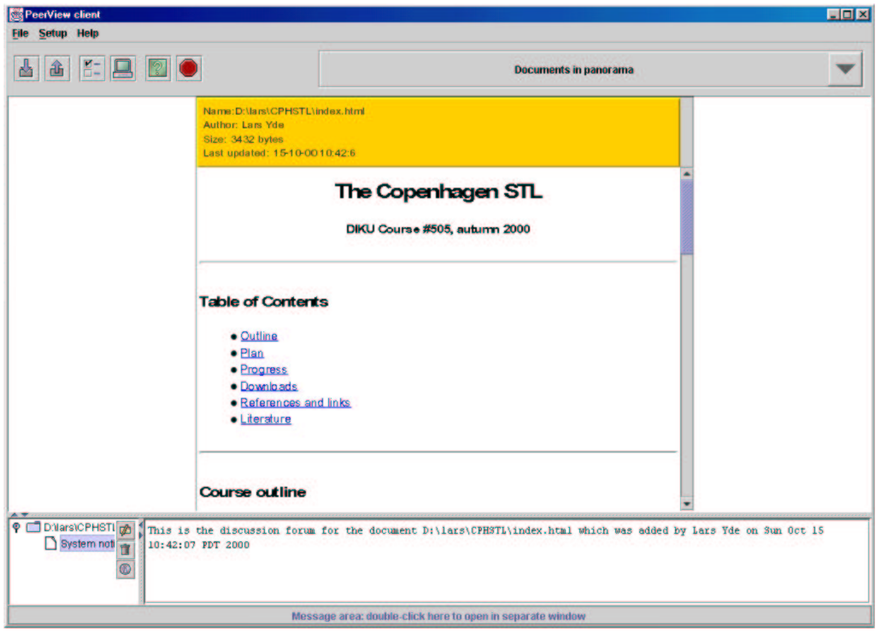
2.6.2 Architecture

PeerView is a client/server application with "fat" clients and a "thin" server, meaning that most system functionality is placed in the client application and relatively little in the server. Each PeerView user is assumed to be running a single instance of the client program on his or her machine, but multiple instances are possible, provided they are running from separate locations (different directories). The server may run on one of the users' machines or it may run on a separate host. Currently, PeerView supports two types of connection: TCP/IP sockets and HTTP. The former is intended for local or closed networks, the latter for use over the Internet, but both may of course be used differently.

2.6.3 The PeerView Client

The client application provides a window onto a set of documents via a zoomable *panorama* [Fig. 2.1, centre] where documents can be arranged in an arbitrary pattern and zoomed to arbitrary scale as well as moved and sized individually. By centring a document, the user can access its *discussion forum* [Fig. 2.1, bottom] and read as well as add contributions to the on-going discussion on that document. The top of the client window is occupied by a selection panel consisting of standard *drop-down menus* and *toolbar buttons*. The drop-down box in the right-hand side of the toolbar panel lists the documents currently displayed in the panorama so that each can be selected by name as well as by navigating the panorama using a mouse.

From the selection panel, the user can choose to add or remove documents, customize the client using a preferences dialog or access and modify the *group directory*, i.e., the list of groups currently available. The directory appears on top of the main window as shown in [Fig. 2.2] where the panorama is zoomed to overview, displaying in this case 32 documents arranged



Name:D:\lars\CPHSTL\index.html
Author: Lars Yde
Size: 3432 bytes
Last updated: 15-10-00 10:42:6

The Copenhagen STL

DIKU Course #505, autumn 2000

Table of Contents

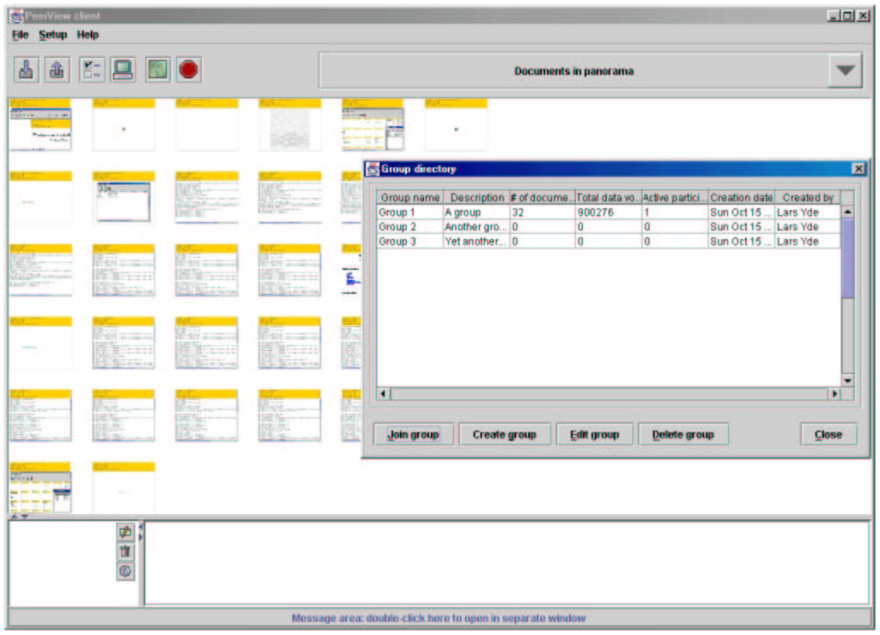
- Outline
- Plan
- Progress
- Downloads
- References and links
- Literature

Course outline

D:\lars\CPHSTL
System noti

```
This is the discussion forum for the document D:\lars\CPHSTL\index.html which was added by Lars Yde on Sun Oct 15
10:42:07 PDT 2000
```

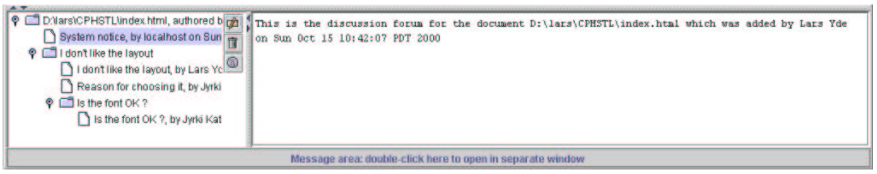
Figure 2.1: The PeerView client window.



Group name	Description	# of docume...	Total data vo...	Active partici...	Creation date	Created by
Group 1	A group	32	900276	1	Sun Oct 15 ...	Lars Yde
Group 2	Another gro...	0	0	0	Sun Oct 15 ...	Lars Yde
Group 3	Yet another...	0	0	0	Sun Oct 15 ...	Lars Yde

Figure 2.2: Panorama overview and group directory.

in a grid pattern which can be customized by the user from the preferences dialog available from the drop-down menus. Each group is listed with its current number of participants, the number of documents it comprises and their total volume in bytes. Using the button panel at the bottom of the group directory box, the user can join, create, edit and delete groups. After joining a group, the other group members, if any, are instructed by the client application to submit their documents so that they can be added to the panorama of the new group member.


D:\lars\CPHSTL\index.html, authored b
System notice, by localhost on Sun
I don't like the layout
I don't like the layout, by Lars Yc
Reason for choosing it, by Jyrki
Is the font OK ?
Is the font OK ?, by Jyrki Kat
Message area: double-click here to open in separate window

```
This is the discussion forum for the document D:\lars\CPHSTL\index.html which was added by Lars Yde
on Sun Oct 15 10:42:07 PDT 2000
```

Figure 2.3: Discussion forum.

The new member's own documents are then broadcast to the other group members so that their panoramas can be similarly updated.

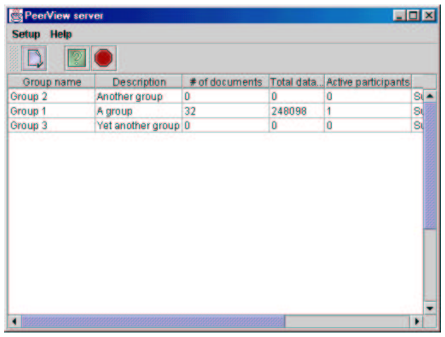
As indicated earlier, centring a document (by double-clicking it) causes the PeerView client to request the discussion forum for that document from the server and, upon receipt, display it in the bottommost section of the main window. From there, the user can select a position in the *topic tree* [Fig. 2.3, left] and using the small vertical toolbar panel choose to compose a new message, delete an existing message that has not yet been responded to, or view the *property sheet* of the selected message.

2.6.4 The PeerView Server

The central purpose of the server is to maintain a database of clients, groups and discussion fora which is reflected in its user interface [see Fig. 2.4]. It consists of little else than a listing of the groups supported, a toolbar and a menu panel from which a few, basic functions (setup and help) can be selected.

2.6.5 Design Rationale

The choice of a client/server architecture was made based on the responsibilities of the server. A peer-to-peer architecture would have been equally feasible but would merely have placed the server functionality in an arbitrary client, resulting in little discernible difference to the user. More significant is the use of a zoomable panorama. As explained earlier, the underlying motivation is to provide overview without loss of content. The user can achieve this by zooming the arrangement of documents to his or her preferred scale, of course, but by combining zooming with a linear setup, an effect similar to that found in work by Eick et al. on the visualization



Group name	Description	# of documents	Total data...	Active participants	
Group 2	Another group	0	0	0	St
Group 1	A group	32	248098	1	St
Group 3	Yet another group	0	0	0	St

Figure 2.4: The PeerView server window.

tool SeeSoft [52, p. 315] can be produced. SeeSoft uses a linear arrangement of multi-coloured columns to render an overview of a collection of code files and their associated statistics (such as time of modification), thus giving users access to large amounts of information at a glance.

On a more theoretical level, the idea of a customizable panorama that can be viewed at arbitrary levels of detail is in keeping with the notion of micro/macro readings as described by Tufte in [55, p. 38 ff.]:

> "...the power of micro/macro designs holds for every type of data display as well as for topographic views and landscape panoramas. Such designs can report immense detail, organizing complexity through multiple and (often) hierarchical layers of contextual reading."

Tufte's focus is on the presentation of cartographical and scientific data but his observations seem to apply equally well here.

2.7 Implementation Overview

The most important considerations in choosing implementation technology for PeerView were: broad user appeal, ease-of-use, high degree of portability, extensible structure and reliability. The Java programming language seemed the obvious choice given these requirements, but implementing all of the planned functionality bottom-up within the set time-frame of four months was almost surely infeasible since implementation was in the hands of a single developer (Lars Yde). The solution lay in using component libraries to implement some of the low-level features such as data distribution and visualization that would otherwise have demanded weeks of development, testing, and fine-tuning to function satisfactorily. The specific choice of components and the reason for choosing them are detailed below.

2.7.1 The User Interface

The PeerView interface is implemented using the Swing classes found in the Java Development Kit (version 1.2.2) [50]. These are light-weight graphical components meaning that they make no (or only very limited) use of operating-system-specific code, thus ensuring maximum portability and a minimum of visual variability across platforms. The zoomable panorama which occupies the centre portion of the client window is built using the Jazz toolkit (version 1.0) [34], which provides primitives for building scenegraphs and scaling text and/or graphics. (A *scenegraph* is a tree structure for organizing graphical objects.) Jazz interacts smoothly with other Java technologies and its structure facilitates future extensions and modifications. In that sense, it allows development time to be shortened without imposing arbitrary restrictions on future development efforts.

2.7.2 Communication and Data Distribution

The PeerView communication infrastructure is implemented using the Java Shared Data Toolkit (JSDT, version 2.0) [36]. It is a relatively low-level toolkit compared to some of the alternatives surveyed, such as NCSA Habanero [28], TANGO Interactive [53], and DreamTeam

[18], all available online and free of charge. However, it has the distinct advantage of being 100% pure Java, meaning that the risk of incompatibility problems — a frequent stumbling block to developers — seemed minimized.

The JSDT requires a server to maintain a directory of clients and groups, and allows all members of a group to communicate and exchange data using an arbitrary number of *channels*. Currently, PeerView uses only a single channel per group, but extending the design later on would be easy as would adding a number of advanced features such as data encryption, access restriction and communication statistics gathering, all of which are supported in part or whole by the JSDT.

As mentioned earlier, PeerView currently supports two communication protocols, namely TCP/IP sockets and HTTP, but can easily be extended with additional protocols since the JSDT design is completely independent of its underlying implementation. Such an extension is, however, a low priority as other, more interesting features (suggested in the previous paragraph) are higher on the agenda.

2.8 Potential Uses for PeerView

PeerView was designed for a specific purpose, namely support of intellectual teamwork, but was deliberately made flexible through simple design and portable technology. It is therefore suited for other uses than that of facilitating intellectual work. Some possible applications of PeerView are listed below. Many of these are inspired by examples found at the Jazz website [34].

Repository inspection: PeerView could be used for the inspection of document repositories such as those maintained by the version management systems that are often used in software development. It is planned to try this in practice in the near future by allowing users to download the PeerView client software from a website and then access a CVS repository using that software (CVS is short for Concurrent Versions System — a popular open source version management system, see [14]).

Web browsing: PeerView can display HTML documents in its present form and can be extended to support hyperlinks in future versions, and could, after such an extension has been implemented, function as an inter-/intranet browser.

Presentation: Documents can be displayed and arranged in PeerView's panorama and subsequently used in a presentation much like conventional slides but in a more manageable way, for example, by projecting the panorama onto a canvas and then using a mouse to navigate while giving the presentation.

Authoring: PeerView could be used as an authoring tool by researchers who could have separate copies of a shared manuscript placed on the panorama and then edit and annotate their version while conferring with others about their copies. This could also be used for commentary and proof-reading and would scale well since the zoomable PeerView panorama need not consume more user screen space as the number of authors increases.

Education: Due to its simple design and graphical interface, PeerView is accessible to most users, including children and some groups of disabled people, and so could be used both for distance learning in geographically dispersed communities and as a support tool in conventional education. One obvious example of the latter is letting visually impaired students study teaching materials at their preferred magnification using the zoomable PeerView panorama.

The above points to some general areas of application, but the number of specific uses is potentially quite large because of PeerView's extensible, open-ended architecture and design.

2.9 Earlier Work

The extensive directory of computer-supported collaborative work and groupware resources at [56] lists a large number of groupware applications for shared editing, conferencing, e-mailing, and other forms of group collaboration. In theory, such functionality can serve the same purpose as PeerView, but in practice, many of these products are designed for different purposes and are thus ill suited for artifact rendering and group review. Two examples are Cybozu Office [15] and Lotus Notes [38], both of which are large suites of tools for collaborative work that facilitate resource sharing and communication. However, their size and complexity makes it impractical to use them for the narrowly defined and specific purpose that PeerView has.

Several academic initiatives have explored areas related to artifact rendering and group review and have produced software that addresses the same underlying problems as PeerView. Examples are the TeamRooms software [46] created for supporting team collaboration and group awareness, and the WORLDS software [60] used for distributed access to one or more repositories of information. Both are similar in architecture to PeerView and are aimed at facilitating collaborative work through object sharing and group communication, but their designs differ from PeerView's in key areas. Both TeamRooms and WORLDS are based on spatial metaphors, i.e., the notion of shared electronic spaces. TeamRooms realizes this idea by letting users define *rooms* which are persistent fora where objects can be shared and communication can take place between the visitors of the room. Each room is equipped with facilities for collaborative work in the form of applets for exchanging information and carrying out other tasks. The TeamRooms user interface is dominated by a panorama onto which the contents of the room (i.e., the shared objects) are projected for inspection and manipulation by all users "present" in the room. An implication of such a design is of course that the set of shared objects can easily occupy more than the visible screen space and TeamRooms addresses this problem by offering a room overview radar that gives an outline, non-scalable bird's eye view of the room (the work on radar overview has been continued at the University of Calgary [24] where TeamRooms was developed).

On the conceptual level, PeerView differs from TeamRooms by being a tool for artifact rendering and group review whereas Teamrooms provides shared electronic spaces for collaborative work. This is the central difference since the conceptual nature of a system usually dictates future design and development efforts. In terms of design, PeerView has fewer features than TeamRooms and consequently a simpler interface. It also addresses the issue of artifact overview differently, namely through its scalable, configurable panorama, but it is similar in its use of object sharing by projection onto a panorama surface. Technologically, the most important difference is that PeerView is written in Java while TeamRooms is a Tcl/Tk application which obviously impacts portability and potential audience since Java is platform-independent and Tcl/Tk is not, although it is widely supported. The WORLDS collaboration environment, which is called *Orbit*, differs from PeerView in much the same way as TeamRooms, but has a more complex architecture and, consequently, more stringent system requirements than both TeamRooms and PeerView.

2.10 Future Plans

Given the open design and implementation of PeerView, we expect little difficulty in integrating new features suggested by future users and ourselves. A handful of useful extensions have already been brought to our attention and are listed below:

- New display managers to allow advanced document formats such as RCS (the format used by CVS [14] for storing the revision history) and streaming video in addition to the existing formats which include plain text, HTML, RTF, GIF and JPG.
- A selection of layout managers, i.e., user configurable program components for automated control of the documents in the client panorama, in addition to the default manager (the grid layout manager) now available.
- An editable and extensible property sheet associated with each document.
- Discussion fora for groups of documents organized by owner, topic, or some other shared characteristic.

The above features are for integration into future releases along with any additional suggestions deemed worthwhile.

The long term plan is to gather feedback about the use of PeerView and from that assess the viability of its philosophy and design. Provided this assessment is favourable, we plan to develop a more ambitious system for real-time artifact rendering and *process rendering*. That is, in addition to an artifact, the organization and history of a project is rendered. The system is to assemble global statistics about the state of a project by collating information and documents gathered from individual clients distributed in a network. Relevant statistics would be the number of documents worked on by each user, the number and position of changes made to those documents and, for code files, profiling and debugging information. That information should then be organized by the server and transmitted around the network to an arbitrary number of consumers who would render it, either visually or by some other medium determined by the preferences of the user. Moreover, the system could offer the review facilities of PeerView along with a set of more advanced features. Ideally, the latter would include "scribbling pads" to communicate in pictures/diagrams in addition to words, audio and video conferencing to allow real-time communication among group members, and playback facilities for observing the evolution of a project over time. Thus, by combining artifact rendering with process rendering, a both spatially and temporally accurate representation of on-going work is given.

2.11 Conclusion

The problems that PeerView seeks to address are not new, but its approach and design are relatively untried. Only user feedback and continued research will tell if such an approach is justified and has enough substance to bear larger initiatives. If so, the potential benefits from a more ambitious project as outlined seem significant. On a more philosophical level, the potential merit of the PeerView approach comes from the fact that it is a response to a real and pervasive problem in software engineering, namely how to get a cognitive grip of an intrinsically intangible and usually highly complex subject matter. In that sense, PeerView can be seen as a small step in the evolution towards CASE tools that can help software engineering attain the maturity of other engineering disciplines and thereby, hopefully, stem the pandemic of failed projects.

Software Availability

The latest PeerView binaries are available online at:

`http://www.diku.dk/research-groups/performance-engineering/PeerView/`

The source code and accompanying documentation will be made available at the above address before the end of this year (2000). Future developments and release plans will also appear there as will related resources. Please feel free to download PeerView and test its usefulness for yourself.

Acknowledgements

This work was supported by the Danish Natural Science Research Council under contract 9801749 (project Performance Engineering).

Chapter 3

PeerView - background and details

The article format places some natural restrictions on style and content, so the level of detail in the above has by necessity been limited. The following paragraphs seek to remedy that by providing the background details on the rationale for the design of PeerView, the reasons for the choice of implementation technologies and architecture, and a discussion of the development process. In the below, PeerView is referred to with its current version number, 1.0, when necessary for the specific point being made or item being discussed. In all other cases, it is simply referred to as PeerView, i.e. without a version number.

3.1 Analysis

The analysis phase of the PeerView 1.0 design consisted mostly in refinement of the ideas that underpinned it, namely support for intellectual work through artifact rendering and group review. For a full-scale system, this would have included empirical research among a target audience to ascertain their preferences and proficiency. Even for a prototype system like PeerView 1.0, field research would have been useful, but I deemed it infeasible within my set timeframe. Naturally, that decision influenced my perspective on the design: PeerView was to be a vehicle for certain ideas, and at the same time be potentially useful to a wide audience to demonstrate the generality, in addition to the feasibility, of those ideas. The implication to me was that PeerView should be simple enough to maintain focus on the core ideas and facilitate wide use, yet comprehensive enough to provide both real functionality for its users and material for future research.

The central issues that had to be resolved first are discussed below in the order they were considered.

The semantics of representation

A central question was of course: what was meant by artifact rendering in practice? What needed clarification, in other words, were the semantics of representation, i.e. what constituted an artifact and which of its constituents should a representation render to provide a useful model for its users.

The answer lay in the fact that any artifact of intellectual work can be represented by a finite string of symbols. The plausibility of this statement follows from its wording: if intellectual work has resulted in the creation of an artifact, then it has been converted from its abstract state ("thought stuff") to material form (a physical artifact). If we assume that all material objects can be exhaustively described using some symbolic formalism, then any artifact can be encoded in a computer readable form since any such formalism can be converted to binary numbers using a binary enumeration of its symbols. This is an abstract formulation, to be sure, but it allows us to focus on the essential aspect of computerized artifacts, namely content. Whether the artifact be text, graphics, audio, or something else entirely, it will always have a string representation. This is one defining property of an artifact, but it is so abstract a definition that it is almost vacuous. Any rendering mechanism would require additional information, i.e. "meta-data", about the artifact such as its medium (text, graphics, audio), its author, possibly its revision history and so on. The exact composition of the meta-data should depend on the type of rendering used: 3D graphics might require different information than 2D graphics, for example.

The form of representation

With a workable, albeit abstract, definition of the artifact concept in place, the next step was to decide on an appropriate form of representation, i.e. the appearance of the artifact rendition. I chose visual rendering as the basis for the interface design, but although this may seem the obvious choice it is not predetermined, at least not if one defines visualization as done in [52, p. 32]:

> "Visualization may be defined as 'the power or process of forming a mental picture or vision of something not actually present to the sight' ... Notice that this definition allows for the use of sensory modalities other than vision, e.g. hearing ... to assist in the formation of mental picture or images."

The reason for my preferring a visual rendering mechanism was the conceptual and technological difficulty I foresaw in implementing a system that supported multiple sensory modalities or merely relied on a single modality other than sight. Also, it seemed overly experimental to base a prototype design on something as relatively untried as auditory, tactile or, in the extreme, olfactory perception!

The visual interface should of course be tailored to needs and expectations of its intended audience, but there were other, more specific concerns. The design chosen would for example have to address two recurring problems for designers of graphical interfaces, namely visual clutter and overcrowdedness, meaning that interfaces can fill up with objects such as windows stacked upon each other until they make up an unmanageable jumble. For systems based on the desktop metaphors there are various solutions such as virtual desktops and multiple desktops on the same machine, but for systems with a potentially large number of objects, such as groupware applications relying on object sharing, this may not suffice. Such has been the experience of Greenberg et al. who in [47] describe the problem as follows:

> "A central concern in information visualisation is how a system can present both global structure (that provides overview and context) and local detail (that reveals information in the user's area of interest)."

They then propose two complementary software solutions to the problem, namely a fisheye view mechanism for use with shared editors so that multiple authors can follow each others work within the same focus, and a magnification lens for use with a graph editor, allowing customized magnification of portions of a graph being edited.

I did not become aware of the above research until after I had begun implementation, but I seem to have followed a similar line of reasoning, namely that the central problem was that of combining local display with global view. This initially led me to 3D graphics as that intuitively seemed the best way to represent a large volume of visual information within the confines of a desktop display. However, the advantages of 3D display, namely increased information volume, flexibility of design and navigation and appeal to human spatial cognition should be weighed against its disadvantages, summarized in [1]:

- Initial confusion and disorientation combined with the additional complexities of the interface and greater freedom of movement. Navigating with six degrees of freedom within an information space as opposed to navigating a set of 2D windows on an information space through panning and zoom controls.
- Large processing overheads are evident when a typically fast computer is reduced to crawl by a complex visualisation. Additional equipment such as a large high resolution monitor and specialized input devices are also a benefit, though by no means a necessity.
- Navigating any 3D environment requires a level of spatial awareness from the user. Navigating within an abstract information space demands more from this skill. Such spaces do not typically conform to our preconceptions of a 3D environment and various expectations such as a notion of “up” and “down” can easily be violated.

Also, it was my experience that 3D graphics displays are more taxing to develop than 2D graphics displays since the former is inherently more complex, thus requiring more resources to implement, debug and maintain. This gap has been closing for some time as freely available component libraries mature (such as OpenGL, see [44]) since these can help lift the burden of implementing a 3D renderer. The point remains, though, that 3D graphics add complexity to a user interface and have the added disadvantages listed above. I therefore decided to use 3D only if I could not formulate a satisfactory 2D alternative.

The challenge was therefore to create a 2D display that could be used for detailed inspection of each element in a set of artifacts yet easily transformed to provide an overview of arbitrary subsets. A scalable desktop seemed to have all the desired characteristics. If one had a 2D surface based on the standard desktop metaphor that could be scaled to arbitrary levels of magnification, one could place a set of artifacts on that desktop and navigate it along the x, y and z axes, thus solving the problem of visual clutter while adhering to GUI conventions known by most users and well supported technologically. In terms of implementation, this solution seemed manageable since scaling a desktop is equivalent to scaling a bitmap which is a well researched graphics transformation [2, p. 78 ff.]. This still left the exact appearance of each artifact on the desktop to be determined, but I considered that a design question and thus left it to be determined at that stage.

Modes of communication

To be a prototype for both the ideas of artifact rendering and group review, PeerView should provide facilities for communicating about its rendering in addition to facilities for displaying it. There are a variety of existing models for supporting group communication, such as mailing lists, USENET news, online chat, email services, and so forth. These provide suitable metaphors for communication facilities in some existing groupware systems such as FirstClass [12] and Lotus Notes [38], and were therefore both tried and conventionalized which made them seem safe, albeit conservative, choices of communication medium. I considered opting for a more experimental solution, but decided to err on the side of caution, bearing in mind the intended audience for PeerView and the time constraints I faced[1].

I reasoned that PeerView 1.0 should at least have facilities for threaded discussions in the manner of a USENET discussion [57] and, preferably, support for online peer-to-peer communication (chat) since this would allow review material to be accumulated and organized in a tried and tested form, and still allow sundry communications to take place more privately, i.e. without the perhaps dissuasive logging that discussion fora entail. Adding more features might contribute to usability, but only at the expense of interface simplicity and development time.

Architecture

Since PeerView was to be used for group review about a set of artifacts which, by implication, would have be shared, it was clear that the system architecture had to be distributed. There are a number of ways to implement such an architecture; some commonly used models are peer-to-peer, client/server and multicast, so a choice still had to be made between the options available. Under ideal circumstances, this choice would be made without consideration for the technologies available; one would simply assume that the required means of implementation were obtainable and design accordingly, i.e. without thought of technological limitations other than the state-of-the-art. However, this is obviously an unrealistic stance since lack of technology support can thwart even the best of designs. It was infeasible for me to build a distributed architecture bottom up within my set timeframe. I therefore decided to only survey component libraries for building communications layers in distributed applications, preferably tailored specifically for collaborative applications. This meant that the implementation language had to be decided on since any communications library was apt to be specifically targeted at a single programming language.

I considered `C++`, Java and SmallTalk as possible implementation languages because they seemed to offer the best mix of extensive library bases, cross-platform support, good development facilities and reasonable run-time performance. Java was chosen over `C++` and SmallTalk because of its inherent cross-platform support and the existence of suitable component libraries for the implementation of PeerView. In fact, the existence of suitable component libraries came to dictate the choice of programming language rather than vice versa. In particular, it quickly became apparent that numerous groupware toolkits [25] existed that were almost all implemented in Java and that more general purpose libraries for collaborative applications, notably the Java Shared Data Toolkit (JSDT) also were available. I chose the latter for PeerView

[1]Section 6 has a more detailed discussion on alternative implementations.

since the toolkits I surveyed all had drawbacks: either they were discontinued, as in the case of Habanero [28], not suited [18] or otherwise unsatisfactory [53]. Furthermore, JSDT had the added advantage of being pure Java and licensed by Sun which minimized the potential for development difficulty such as interfacing different technologies.

JSDT can be used to implement client/server and multicast architectures, but the server need only be assigned minimal responsibilities such as maintaining a registry of clients so it could be used to construct an architecture that effectively would function in a peer-to-peer manner. I was therefore not concerned about limiting myself to a particular architecture prematurely since JSDT seemed flexible enough to accomodate potential revisions later on in development.

3.2 Design

A commonly used distinction is that between the interface of an application, meaning the graphical components that the user interacts with, and the functionality, meaning the actions that the user can carry out by interacting with the interface. This is not the only possible division, of course, but it reflects the actual design process, so the below discussion reflects that view.

3.2.1 Interface design

Java was chosen as the implementation language for PeerView during the analysis phase, and this in part dictated the design of the GUI since the look-and-feel (as the graphical appearance and response pattern of an interface are sometimes called) of the interface widgets (i.e. windows, buttons, scrollbars etc.) should be that provided by the Java AWT or Swing interface classes. Choosing a different look-and-feel did not seem sensible as the Java classes implement all common interface primitives and can be extended arbitrarily. I chose the Swing class set which is Java's lightweight, i.e. platform independent, GUI library rather than the AWT which shipped with early releases of Java, but is now being phased out.

The specific layout of the PeerView interface should both cater to its intended audience by being accessible and intuitive or at least adhering to convention, yet at the same time fulfill its other purpose, namely that of clearly demonstrating the ideas it is built on. The most important design goal could therefore be stated as providing the user with access to the functionality required for carrying out a particular task without distracting from that task, preferably in an aesthetically pleasing manner. In practical terms, this meant sparse displays, i.e. windows and boxes which are not densely packed with widgets, and sensible layout, e.g. associating selection controls with the display elements they controlled. This is supported by Schneiderman [49, p. 318] who reports:

> "Crowded displays are more difficult to scan, especially for novice users. In a NASA study of space-shuttle displays, sparsely filled screens with approximately 70-percent blanks were searched in an average of 3.4 seconds, but more densely packed screens with approximately 30-percent blanks took an average of 5.0 seconds ... This study also demonstrated that functionally grouped displays yielded shorter search times."

It is also in keeping with the heuristics formulated by Nielsen in [43] who advocates:

> “The ideal is to present exactly the information the user needs — and no more — at exactly the time and place where it is needed. Information that will be used together should be displayed close together ...” [43, p. 116]

By aesthetically pleasing I did not mean merely according to my personal preferences, but in the sense of consistent use of simple and tasteful visual presentation to further usability. This issue is discussed at length by Nielsen who, among other things, offers the following aphoristic advice on colour, information density, and graphic design, respectively:

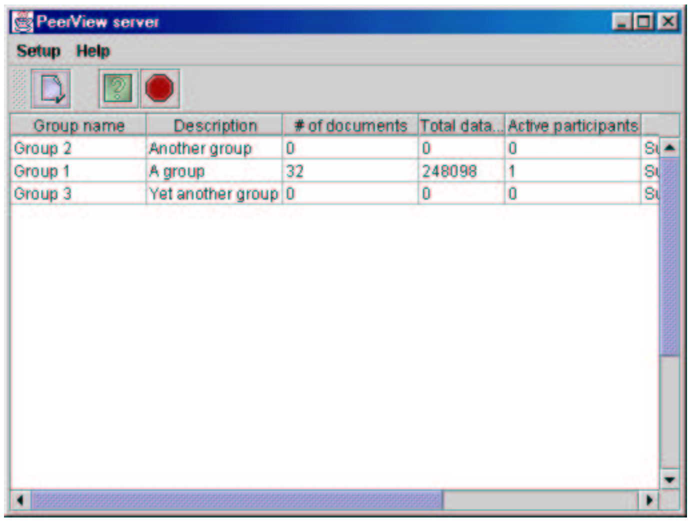
> “Don’t overdo it. An interface should not look like an angry fruit salad of wildly contrasting, highly saturated colors. It is better to limit the design to a small number of consistently applied colors.” [43, p. 119]

> “The ‘less is more’ rule does not just apply to the information contents of the screen but also to the choice of features and interaction mechanisms for a program. A common design pitfall is to believe that ‘by providing lots of options and several ways of doing things, we can satisfy everybody’. Unfortunately, you do have to make the hard choices yourself.” [43, p. 121]

> “Screen layouts should use the gestalt rules for human perception to increase the users’ understanding of relationsships between the dialogue. These rules say that things are seen as belonging together, as a group, or as a unit, if they are close together, are enclosed by lines or boxes, move or change together, or look alike with respect to shape color, size or topography.” [43, p. 117]

Having settled on the guidelines for design, I immediately began translating it into reality using a rapid prototyping tool. Some software engineering pundits recommend designing displays on paper before committing them to electronic form, but with the graphical design tools available today, I find this approach unwieldy. I started by designing the PeerView client main window since this was the “control center” of the application, so to speak. The result is shown in one of the figures used in chapter 2.4, namely Figure 2.1. The window is divided into three parts: the control section at the top, the middle portion containing the scalable desktop and a bottom portion containing the discussion forum, which in turn is divided into a navigation frame and a content frame. The sizes of these elements, except the control Section, can be adjusted using the divider bars[2] that separate them. This layout was chosen because it correlated with the conventions of GUI design by having control at the top, work area below and text input areas at the bottom. I would have preferred to have the quality of this design verified through suitability testing in a realistic setting, but had neither the time nor the means to do so, which was another reason for making conservative design choices. The drop-down menus and toolbar in the control Section were populated by items reflecting the functionality I could foresee from the results of my analysis, and were therefore likely to change as design, and perhaps even implementation, proceeded. The same applied to the pop-up menu that could be activated

[2]Vertical and horizontal bars with a ridged surface and arrow points at one end.



Group name	Description	# of documents	Total data...	Active participants	
Group 2	Another group	0	0	0	Su
Group 1	A group	32	248098	1	Su
Group 3	Yet another group	0	0	0	Su

Figure 3.1: The PeerView server main window.

by right clicking on an artifact, i.e. I put in the menu items I believed necessary to have a functioning product in place and left room for extensions if need be. The list box in the right hand side of the toolbar contains a listing of artifacts, i.e. documents, shown in the panorama organized by author. By selecting a title from the box, the user can center the panorama on the corresponding artifact. This form of iterative development may seem meandering, but it is, in my opinion, more realistic than a rigid, linear design process that leaves little room for exploration.

The PeerView server window, shown in Figure 3.1, reflects its purpose, namely to serve as registry and nexus for communication between clients. Consequently, it is kept simple with only a control Section at its top and a listing of groups below that.

3.2.2 Functionality design

As I had found suitable technologies for implementation during the analysis phase, some of the design issues had already been settled implicitly. Specifically, the Java Shared Data Toolkit implements the concept of *client*s partaking in *session*s using *channel*s as lines of communication. Using that technology therefore meant designing functionality accordingly, i.e. supporting similar concepts. The PeerView client application was therefore designed to operate in groups, which were to be implemented as JSDT sessions ("group" was chosen because it connoted a more permanent affiliation than "session"), and communicate using JSDT channels. This was the bare minimum of communications infrastructure, but I decided to limit the initial design to that to keep implementation manageable although the JSDT contained other facilities that could have benefited the design.

The functionality of the scalable desktop was more of an open question as it depended largely on what could be implemented in the time available. The Jazz library [34] proved helpful in answering that question. Jazz is a library of classes specifically targeted at creating scalable (or zoomable, if you will) interfaces, appropriately named ZUIs, and by using that library, I could manage considerably more sophisticated designs than if I had had to implement everything bottom up. Jazz provides primitives for constructing and managing a scenegraph [8] which is a data structure for organizing objects in a 2D or 3D environment and it implements a fast scaling transform for graphical objects. Furthermore, it can be used with the standard Swing document and bitmap classes and all in all, this seemed to promise a shortened development phase by allowing me to use component technology for both the scalable desktop itself and the artifacts to be rendered on it. In keeping with the conservative design choices made earlier, I decided to limit desktop functionality to the essential operations, namely translation[3] of individual artifacts, scaling of the entire desktop and automatic layout of all artifacts. Later, after having implemented and tested these features, I realized the need for additional ones, but given the pop-up menu interface to the desktop functionality, it was straightforward to add more features by simply implementing the required functions and adding items to the menu. Specifically, it proved useful to be able to "pin" artifacts to the desktop, i.e. fix their positions so that they were not repositioned once a rearrange operation was carried out, so this and a handful of other features were added to the desktop menu before releasing the beta version.

To put these ideas of functionality into more concrete form, I wrote a series of use cases, i.e. simple narratives describing typical scenarios of use, as suggested by [33]. Writing use cases after having outlined functionality and interface was somewhat unorthodox as standard procedure is to begin by writing use cases, preferably in collaboration with the intended users of the system. However, this was not a standard development project in that the problem domain and audience were abstract rather than concrete, so it did not make sense to me to follow standard procedure. That is, in the absence of a concrete problem domain, one has to flesh out one's ideas before use case narratives can be written about them. This could be either in the abstract, i.e. in the mind, or in more concrete form through design which is what I opted for in this case. Appendix A contains a use case diagram and brief textual descriptions of each case.

3.2.3 Class design

With both interface and functionality largely decided upon, I could outline the class structure of PeerView. The resultant class diagram is shown in Appendix A, and the following describes the rationale behind the design.

The design was kept as simple as possible to avoid complications and allow easy modifications at later stages. The classes that make up the design are divided into interface, entity, and control classes. The *interface classes* are those implementing the GUI, the *entity classes* are those that contain and manage the application data structures without performing any but minor operations on them, and *control classes* are those that mediate between model and interface by modifying the former and updating the latter in response to user interaction. This division is a very simple object pattern, but is nevertheless not adhered to with absolute rigour as flexibility of design had priority at all times. For example, the `Constants` hierarchy in Diagram A.2 is an appendage to the model classes in that it does not represent a data structure, but a repository of constants for use by one or more other classes. This allows constants to stored centrally, accessed through get/set method pairs, and appear where used in the source code as meaningfully phrased function calls rather than string literals. For more

[3]Movement along the x or y axis.

on this, see the source code in Appendix C or Section 3.3. Also, most of the control code is placed in central control classes, namely `ClientManager` and `ServerManager`, and almost all the entity data in central entity classes, namely `ClientData` and `ServerData`. This was done because the interface design clearly indicated that most data structures would be either read from or written to by more than one interface class via the appropriate control class. There was therefore little need to complicate the design by having multiple entity/control pairs (one for each interface class in the extreme case).

The many classes that implement *listener* interfaces are event handlers. These were added as implementation progressed rather than at design time, for although event handling is the standard notification mechanism in Java and the need for handlers therefore was obvious, the exact number and interfaces were difficult to determine at this stage. An attempt to incorporate handlers in the early design was therefore apt to be inadequate. The decision to add event handlers only during implementation seemed unproblematic since handlers are highly standardized, and can be nested in other classes, so I deemed the risk of this causing unplanned structural design change to be minimal.

3.3 Implementation

Implementation was done over a 6 month period from the middle of May 2000 to November 2000 using the VisualAge for Java 3.0 IDE (Integrated Development Environment) [58]. As for implementation methodology, I decided to proceed iteratively and by small increments, i.e. frequent builds[4] and application tests. This may seem an obvious approach, given that it is probably how most small scale projects are carried out, but there are alternatives. In [42], Mills et al. discusses *cleanroom* development where developers are forbidden to test and execute their code while writing it, and tests are based on statistical analysis. The method is directed towards projects with no or low fault tolerance, i.e. projects where the code produced must have a low error to KLOC (Kilo-Line-Of-Code) ratio, and is intended for use by teams of developers as it relies heavily on peer review. A different approach that seems more attuned to development of applications like PeerView is *Extreme Programming* [6] which supports the notion of small increments. It also advocates programming in pairs as a method of ensuring continuous peer review, so I could obviously not adopt extreme programming in toto. Instead, I considered its recommendations as validation of my personal experience from earlier projects which was that small increments and constant feedback stimulates motivation and helps reduce the error rate.

The order in which design features were added to the framework produced during the design phase[5] was determined by how essential they were to the end user, or, to put it differently, how indispensable they were to the application. For example, it would be possible for an end user to make sensible use of PeerView if it provided nothing more than a client application with a scalable desktop and functionality for adding artifacts to it. It would not be possible if that user had only a working server application with a fully functional communications system but nothing else. This approach was chosen because it ensured that a usable, if amputated, system would always be in place for demonstration and preliminary releases if the need arose, and because it allowed realistic interface and functionality testing to take

[4]A build is the compiling and linking of all components making up an application.

[5]The interface classes created during GUI design and the skeleton code for entity and control classes.

place during development. The major features are discussed below in the order they were implemented. A major feature in this context is one whose implementation is non-trivial. An example of the opposite, a trivial feature, could be transfer of string literals from one list box to another, or reading the contents of a set of files.

3.3.1 The scalable desktop

The scalable desktop is implemented by the class `ClientPanorama`. The term "panorama" is used throughout the PeerView implementation and documentation rather than the phrase "scalable desktop" used during design, because I reasoned that the idea of a panorama window providing overview of a vista had more intuitive appeal to the average user than the more technical notion of a scalable desktop. `ClientPanorama` is derived from the Jazz library class `ZCanvas` which implements a simple scenegraph structure, a drawing surface and a viewport camera (shown in Figure 3.2).

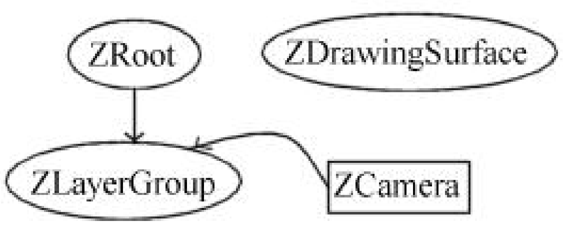


Figure 3.2: The `ZCanvas` scenegraph.

A scenegraph is a collection of hierarchically organized nodes that specify the structure of the scene that is rendered on a users's display. The nodes can be either *leaf nodes* which have no descendants or inner nodes, so called *group nodes*, from which other group nodes or leaf nodes can descend. Operations applied to a group node affects not only that, but also its descendants which allows operations to be applied in a flexible manner to arbitrarily sized subsets of the scenegraph, including the entire scenegraph if operations are applied to the unique root node. Group nodes can be categorized by the types of operations they support and Jazz includes such group types as `ZTransformGroup`, `ZStickyGroup` and `ZInvisibleGroup` which can be used to translate, scale and rotate their descendants, to make them appear at a fixed magnification and to be rendered invisible in the scene, respectively. A separate type of node is the *camera node* which can be attached to the scenegraph at an arbitrary position and used to render an image of the corresponding scene at that position. This means that one can place multiple cameras in a scenegraph and have different scene renderings produced without making changes to the graph itself.

The `ZCanvas` scenegraph contains the minimum of objects required to render a panorama, namely a `ZRoot` object with one child, a `ZLayerGroup` object which groups the set of nodes descended from it. Jazz has several types of such group nodes, most of which allows one to apply an operation to all nodes descended from them by applying that operation to the group node. The `ZCamera` node represents a viewport onto the nodes in the scenegraph and the `ZDrawingSurface` the surface onto which the camera view is projected. Artifacts are added to a `ClientPanorama` object by inserting a `ZVisualLeaf` node under the `ZLayerGroup` of the `ClientPanorama` (or rather, its `ZCanvas` super class). A `ZVisualLeaf` is a leaf node in the scenegraph which has a visible component that can be rendered in the scenegraph's `ZDrawingSurface`. An artifact is translated by adding a `ZTransformGroup` node above the

`ZVisualLeaf` corresponding to that artifact and then applying a translation operation to that `ZTransformGroup`. Before adding more features to the panorama class, I implemented the basic functionality of the `AddFiles` and `RemoveFiles` classes so that artifacts could be placed on and removed from the panorama. This was in keeping with the overall aim of adding functionality in the order of usefulness, but mostly a practical concern since it would be difficult to test operations on the panorama if there was no way of inserting objects into it.

Scaling

The entire panorama can be scaled, but not individual artifacts. The reason for not allowing scaling of a single artifact is not only the overarching decision to maintain simplicity of design in the initial version of PeerView, but also that such a feature could easily be counterproductive. If individual artifacts could be scaled, the resulting panorama would be a mosaic of objects at different positions along the z-axis. Translating an object would then become a potentially user-unfriendly operation since a translation by Δ of an object o at z-coordinate α along either the x or y axis is transformed to a translation by $\frac{\Delta}{\alpha}$ in the `ZDrawingSurface` rendering that is visible to the user. The user would then experience differing responses to identical inputs as an indication to translate an object o at z-coordinate α by Δ would appear to move o further than if another object o' at z-coordinate $n * \alpha$, where $n > 1$, was translated by the same amount.

The scaling from magnification Z to the target magnification Z' is done by interpolating between the two values over a period of length l, where l is user adjustable using the preferences Section of the PeerView client application. The interpolation follows a slow-in-slow-out trajectory and the display is updated after each step, so to the user a scaling operation appears as a smooth animation from Z to Z' that begins and ends with a gradual slowdown rather than an abrupt halt.

Performance

When implementing the translation and scaling operations, it became clear that poor runtime performance would be very detrimental to the usability of PeerView. That is, if those core operations could not be executed smoothly and with little delay on the average system, then users were likely to be exasperated which is obviously not conducive to end user satisfaction. The default behaviour of the Jazz renderer is to render objects on the drawing surface at one of three levels of detail: low, medium or high. These settings of course reflect a trade-off between quality and speed of rendering. I was aware of this at the outset, i.e. when beginning implementation and so had expected that quality could simply be reduced during critical operations to increase the speed at which they were carried out. However, this was not sufficient, for although performance was satisfactory on my system which must today be considered an average configuration PC, it was too sluggish on other systems (such as thin/dumb terminals) that for example did not have dedicated terminal hardware for graphics display, but instead received images from a server through a local network. I therefore modified the renderer so that objects were rendered as filled grey rectangles when translation and scaling operations were carried out. This improved runtime performance significantly, but not enough for all systems so there are still minimum system requirements when using PeerView. However, given the hardware of the average user today, I do not expect this to be a serious problem.

Layout

The panorama should support automatic layout. This was decided during design since it is a feature known from other desktop GUIs that can help the user restore order in a cluttered display, cf. for example the "Arrange" item on the desktop pop-up menu of the MS Windows platform. Layout is performed by a *layout manager* which is a class that accepts a set of nodes and then arranges them using a layout algorithm that may be arbitrarily sophisticated. Currently, PeerView 1.0 provides only the `GridPanoramaLayoutManager` class which is derived from the abstract class `ClientPanoramaLayoutManager` that forms the root of the so far small layout manager class hierarchy. A `GridPanoramaLayoutManager` arranges a set of artifacts in a square grid pattern where spacing is determined by the largest of those artifacts[6]. The user may specify the type of layout manager to be used from the preferences dialog of the client application, but as mentioned, only grid layout is currently available. The layout manager is called when the user inserts artifacts into the panorama and when she selects the "Arrange" item from the panorama pop-up menu. I chose the grid layout algorithm because it was simple to implement and the default layout in most other desktop GUIs. There is no inherent limitation to the level of complexity of the layout manager, though, and one could envision managers that laid out according to semantic criteria such as content, type (text, graphics, etc.) and size in addition of course to the wide variety of geometric layouts that can be conceived. So far, however, I have not found the time to implement any of these, but since a layout manager is simply a class derived from `ClientPanoramaLayoutManager` that adheres to a particular interface and accepts a set of nodes, it should be possible for other developers to add such managers to PeerView 1.0, if desired.

Additional features

After experimenting with the panorama, I found out that it would be useful to be able to resize artifacts and to fix their position, i.e. "pin" them to the panorama surface, so to speak. This would give the user an acceptable degree of freedom in customizing the display without increasing development time significantly. I therefore introduced a resize feature to the `ClientPanorama` class and a corresponding item to the artifact pop-up menu. The operation is carried out by dragging a corner of the artifact in question to the desired position and then left-clicking the mouse. The artifact retains its size until removed or resized again. The ability to fix an artifact at a given position was implemented by maintaining a lock for each artifact (implemented as a boolean data member) and then excluding locked artifacts from any layout operations. The lock can be toggled from the artifact pop-up menu which also allows the user to toggle the default lock setting for any new documents added to the panorama. Both the resize and the lock feature necessitated changes to some of the method definitions in the `GridPanoramaLayoutManager` class.

3.3.2 Message and progress bar

At the very bottom of the client application main window there is a small rectangular area which is used for displaying error and status messages and for indicating progress for operations of some duration. This area, called the message area or message bar, can be double-clicked to

[6]This method of spacing was introduced after implementing the resize feature, see Section 3.3.1

expand into a window containing a log of previously displayed messages. Implementation of the message area was mostly straightforward, but one aspect was non-trivial, namely how to make the progress bar update and display and at the same time perform the operation whose progress the bar was indicating. I tried first to have the code implementing the operation force the progress bar to update at each step of execution, but I learned from a newsgroup conversation that this approach reportedly was grossly inefficient. I decided to rely on this information and implemented the message area as recommended, namely by executing the code for a task whose progress was to be indicated in a separate thread. That code will then modify the progress bar which runs in the event-handling thread[7] and is therefore automatically updated by the system when that thread is executed.

Because of Java's platform independence, the issue of multi-threading is made somewhat more complicated than it appears from the above. Different operating systems implement different multi-tasking policies which affects the way in which threads are allotted processor resources. Notably, the UNIX and Solaris operating systems use *preemptive scheduling* where executing threads are only taken out of their running state when they *yield* control or are forced out because a thread with higher priority requests execution. The Windows NT operating system, however, uses *time slicing* as its multi-tasking control mechanism where processes such as threads are given a fixed size time-slice in which to execute. When that time-slice expires, the thread is put out of its running state and the next contender put into its running state instead, regardless of whether the first thread yielded control or not. In practice, this means that on systems that use preemptive scheduling, the threads started by PeerView may be preempted, i.e. put out of their running state, by unrelated processes that happen to have higher priority and be executing on the system in question at the same time. This will sometimes produce noticeable delays in updating the PeerView interface which the user will most likely experience as erratic changes in the length of the progress bar. The code for launching tasks in separate threads is in the code listing for the `ClientApplication` class in section C.2 (e.g. the inner class `RemoveDistributedDocumentsFromPanorama`).

3.3.3 Communications layer

Before the group management and discussion forum functionality could be implemented, the basic functionality for message and data exchange had to be in place. It is perhaps debatable whether this was in accordance with the decision to add features in the order of usefulness to the end user, but without means of transmitting group and message data it would be difficult to test the components responsible for those types of data. To create a communications layer, I had to add some functionality to the server application skeleton code produced during design. Specifically, the server is responsible for maintaining what the JSDT documentation refers to as a *registry*, i.e. a small database listing the clients and sessions registered with that server. The server also maintains a concurrently updated database of group information, messages and discussion fora, but most of that functionality was added later when both the group directory and the discussion forum components which generate that data had been implemented. Initially, only the methods for creating and destroying a registry and for opening channels of communication was implemented on the server side. On the client side, code was written for methods in the `ClientManager` class to enable it to connect to the server, to join

[7] Java Swing events are always handled by the thread which started the application generating those events.

a session (which, cf. Section 3.2.2, corresponds to a group in PeerView group), and to send and receive data through communications channels.

3.3.4 Group directory

The group directory is shown in Figure 2.2. It provides a listing of available groups and functionality for joining a group, adding or editing one or deleting one from the listing. The interface had been constructed during the design phase so what remained was implementing the appropriate functionality and associating it with the push buttons at the bottom of the group directory dialog box. The add, edit and delete operations were "database primitives" in the sense of being the standard operations for simple interactive data structures which typically provide the ability to insert and delete elements as well as replace existing elements with new versions. Implementing these as operations on the local data structure, i.e. the one whose contents are displayed in the listing above the panel of buttons was therefore straightforward. However, after each operation, that local data structure is submitted to the server which maintains the master copy of the group listing. The server then updates the master to reflect the changes in the local structure and subsequently resubmits the master to all client currently connected to the server. The data structure is typically quite small so transmitting it from client to server in its entirety does not add significant overhead compared to just transmitting a notification of the operation performed and is simpler to implement and extend (new operations can be added without having to introduce a new type of notification, for example).

The "Join group" operation is more composite than the data management operations discussed above. To join a group, the following must be executed, assuming that a group was actually selected when the user opted to join:

1. If the user is currently member of another group, then:
    (a) remove (the copies of) all other group members' artifacts from the user's panorama;
    (b) notify the server that the user has left so that the master copy of the group directory can be updated to reflect the change in number of group members and total data volume;
    (c) notify the other group members that the user has left so that they can remove (their copies of) the user's artifacts from their panoramas.
2. Join the selected group and then
    (a) broadcast copies of all the user's artifacts to all other group members;
    (b) request copies of all other group members' artifacts and add them to the user's panorama when received.

Minor details have been omitted from this definition; they appear from the source code in Appendix C. The operation was implemented so that the client leaving a group need only remove its copies of other group members' artifacts and then actually leave the JSDT session corresponding to the group. The server will then be automatically notified since it monitors all channels of all sessions for leave and join operations. Upon notification, the server will notify all other group members that the client has left the group and they will then update their panoramas accordingly. The server also changes the master copy of the group directory listing and broadcasts the updated copy to all active clients.

The group listing master copy maintained by the server is written to disk when the server exits and read again when it is next started.

3.3.5 Discussion forum

The discussion forum has two components: the *topic tree* which is a collapsible tree structure that organizes hierarchically a set of nested topics, and a message pane (also referred to as the message area when addressing the user since "pane" might not make intuitive sense), where the messages selected from the topic tree are displayed. Each time a user selects an item from the tree that corresponds to a message added by herself or another user, the client application requests that message from the server which maintains a database of all discussions and messages. The server responds to the request by sending a `Message` object which is a simple data structure containing the message text and relevant meta-data such as author name, date and time of composition, etc.

The server database of messages and discussions is written to disk when the server exits and read again when it is next started. The storage format is plain text, which implies that compression might reduce the size significantly. I found it unlikely, however, that the size would become unmanageable great in realistic use scenarios if the data was stored in its uncompressed form and that therefore became the storage format for all such plain text files maintained by the server.

3.3.6 Artifact updating

To provide an accurate representation of a set of artifacts, the PeerView client must ensure that the objects placed in the panorama are updated at regular intervals. The decision to make the exact interval user adjustable was made during design as this affected the interface, but the specific method of updating was left undecided since I reasoned that choosing the best method would require some experimentation which required a code scaffolding, i.e. there had to be a panorama and artifacts for experiments on them to take place.

A naïve and failsafe method of updating artifacts can be formulated using pseudo-code:

```
1.  for each artifact $\alpha$
1.1.  if ( ( changed( $\alpha$ ) == true ) AND ( group( owner( $\alpha$ ) ) != null ) )
1.1.1.  broadcast( $\alpha$ , group( owner( $\alpha$ ) ) )
2.  goto 1
```

The functions called in the above are assumed to be defined elsewhere. Function `changed(` α `)` returns `true` if the artifact α has been changed since the function was last called with α as argument. Function `owner(` α `)` returns a unique identification of the owner of the artifact α, and function `group( id )` returns a reference to the group to which the user corresponding to `id` belongs, or `null` if he or she does not belong to a group. The function `broadcast(` α`, g )` broadcasts a copy of the artifact α to every member of the group except `owner(` α `). g.`

Since PeerView 1.0 supports both text and bitmap artifacts, this is a potentially very inefficient and costly method of updating as the size of a typical bitmap image at resolution

1024x768x24 may be in excess of 3Mb, depending on the storage format. An alternative is to transmit only the difference between an artifact before and after a modification, as outlined below:

```
1.  for each artifact $\alpha$
1.1.  if ( ( changed( $\alpha$ ) == true ) AND ( group( owner( $\alpha$ ) ) != null ) )
1.1.1.  broadcast( diff( $\alpha$ ) , group( owner( $\alpha$ ) ) )
2.  goto 1
```

The function `diff` computes and returns the difference between the artifact α before and after the most recent change was made. Each client that received the difference data d would then execute the following:

```
1.  $\alpha$ = combine( $\alpha$, diff( $\alpha$ ) )
```

The function `combine` computes and returns the artifact α' that results from merging the artifact α with the difference returned by `diff`.

The runtime performance of this method can be enhanced by compressing the difference packages sent between clients, and it can be further improved by adjusting the quality (and thus time complexity) of compression to match the processing power to network bandwidth ratio of the system it is executed on. Such adjustment can be either automatic or user definable, depending on the implementation effort one is willing to invest.

Simple back-of-the-envelope calculations show that an implementation of the difference algorithm using for example the DEFLATE algorithm [16] for compression will outperform the naïve algorithm by an order of magnitude or more:

Notation:

d the average size of a document in bytes

s the average size in bytes of the difference between a document before and after a modification.

u the number of update operations pr. sec.

c the compression factor.

N the naïve algorithm bandwidth consumption pr. sec.

C the compressed algorithm bandwidth consumption pr. sec.

Assume:

$d = n$
$s = \frac{n}{10}$
$c = 4$
$u = 3$

Then:

$$N = d * u = n * 3 = 3n$$

$$C = \frac{s * u}{c} = \frac{\frac{n}{10} * 3}{4} = 0.075n$$

and if for example $n = 10\text{KB} = 10 * 1024$ bytes $= 10.240$ bytes then $N = 30.720$ bytes and $C = 768$ bytes.

Although these were loosely based calculations, they convinced me that experiments were unnecessary: the difference algorithm was the best choice, since its time/space tradeoff was best by a wide margin.

However, my first attempt at implementing a difference algorithm was hurried and not well enough thought through as a discussion with Jyrki Katajainen made clear. I had gone ahead with my initial idea for a difference algorithm which consisted in computing the difference between two byte arrays representing a document before and after a change by performing a logical XOR operation between the overlapping portions and appending whatever remained (which I surmised was likely to be neglible) to the difference which was also stored in an array. This difference array would then consist of long strings of 0's and 1's indicating areas of change and areas that were unaffected by the modifications made since the previous update. I then used the Java `Deflater` class [50] to compress the difference array before it was broadcast to other clients who in turn used the Java `Inflater` class [50] to decompress the difference array. They then combined the difference array with their old version of the document it was associated with by performing the same XOR operation as was used to derive the difference. Because of the typical contents of the difference array (long sequences of 0's and 1's), the compression ratios were quite impressive and the entire arrangement seemed to simple and symmetric that I conveniently forgot to think actual usage scenarios through. The truth of the matter is that this XOR difference approach is only effective if the size of the array storing the document before the change is exactly the same as the size of the array storing the document after the change. Only then will the difference array contain only the difference between the two input arrays, punctuated by long sequences of identical symbols and thus yielding the favourable compression ratios compared to transmitting the entire document [8]I was after. If, on the other hand, the change to a document resulted in a change to its size then the difference array would contain a sequence of identical symbols of length equal to the distance from the beginning of the document to the beginning of the changed portion. The remainder of the difference array, i.e. the portion corresponding to the beginning of the change in the document to the end of the document, would be an XOR combination of the document before and after it was changed, and because of its change in size, those two versions would be shifted relative to each other. If we assume that, on the average, a document will be changed at position $\frac{n}{2}$ where n is the length of the document (and we ignore the size of the change itself), the difference array can only be compressed to about half the size that the original document could since it is, on average, half as "random", to put it informally. An example might make this clearer:

[8]Typical compression schemes will reduce a long sequence of identical symbols to a representation consisting of the symbol being repeated throughout the sequence and the number of times it is repeated, i.e. the sequence length.

	Modification without change of size	Modification with change of size
Original array	0001100110011111110000111111	0001100110011111110000111111
Modified array	0001100110011000110000111111	0001100110001
Difference array	0000000000000111000000000000	0000000000010111110000111111

As the above suggests, the XOR difference array algorithm was far from optimal in that it only works satisfactorily if a change to a document does not result in a change to its size which is, of course, an overly restrictive limitation. It is, however, the algorithm used in the current implementation of PeerView as I have not had time to implement a better alternative. Some such alternatives are described in [31], where Hunt et al. survey different *delta algorithms* as algorithms for computing the difference between successive versions of a document are typically called in available research literature. They find that the traditional `diff` delta compressor found on the UNIX operating system is outperformed by more recent algorithms based on Ziv-Lempel techniques (such as the DEFLATE algorithm used in PeerView) and they present an alternative that they call *vdelta* which performs better, both in terms of compression ratios and in terms of compression/decompression times, than `diff`, `diff` combined with `gzip` compression and `bdiff`, a successor to `diff`.

The vdelta algorithm therefore looks like a promising candidate for use in PeerView. Indeed, the authors suggest that the algorithm is well suited for use in such applications [31, p. 10]. I have, however, not given implementation of an improved algorithm top priority since the current version functions satisfactorily: typical compression ratios are better than 1:15 (averaged over text and graphics) so on all but very large documents that are changed with short intervals, it performs quite well. If the system was to be developed further, however, the algorithm would have to be replaced since it consumes an unneccesary amount of bandwidth as the number of users increases.

Chapter 4

Experimental evaluation of PeerView

4.1 Introduction

Evaluating groupware is difficult because it combines the traditional problems involved in evaluating single user system with the complications introduced by scaling to multiple users [3, 45]. Among those complications are both practical concerns, such as recruiting and compensating users, planning and conducting experiments and scheduling them as well as methodological concerns such as how to best gather and use experimental data. This makes it difficult to formulate a suitable procedure for evaluating groupware. In fact, Baker et al. [3] claim that no cost-effective techniques to that end exist and that one is therefore consigned to adapting existing, single-user evaluation techniques to a groupware context.

I opted for a controlled experiment with a small group of participants who were asked to carry out a set of tasks using PeerView and subsequently to fill out a questionnaire and comment on their experience using the software. The reason for making this choice is discussed in further details in Section 4.3. The experiment itself was recorded on audio tape[1] and combined with the written feedback from the three participants, it yielded substantial information on the usability of PeerView.

The below discussion is based on the materials used in the course of the experiment, all of which can be found in Appendix B.

4.2 The experiment

The experimental evaluation of PeerView was carried out on March 14th, 2001. The setting was a room in a Copenhagen computer café equipped with four medium-range PC systems running Microsoft Windows 98. The machines were fitted with 17" displays and Pentium II processors and were connected by a local area network. On one of the machines, the PeerView server software had been installed in advance since I did not consider the installation and configuration relevant to this end user experiment whose primary focus was to be on the

[1]Circumstances did not allow for video recording, but as appears from the transscript (a written reproduction of the dialogue on the audio tape) in Appendix B.2, the participants' comments and questions were quite interesting in themselves and gave enough information to deduce many of their actions.

client side software. The other terminals contained only the software installed by the café proprietors, i.e. mostly communications software and computer games.

The three participants were a 27 year old woman (referred to as "Rikke" in the transscript) and two men aged 26 ("Thomas") and 28 ("Paul"), respectively. They all had good working knowledge of how to use computers on a day-to-day basis, but had no super-user skills or prior experience with either groupware or zoomable interfaces. They were each assigned a PC and handed a task list (reproduced in Appendix B.2) which they were asked to read and proceed with. The tasks where linearly structured with each being dependent on its predecessor on the list, but required use of almost all the functionality in PeerView and therefore constituted a thorough walkthrough of the system. This setup was chosen because it allowed me to focus on the participants' behaviour and performance rather than on explaining the intent behind a set of open tasks with no specification of how to perform them.

The first task on the list was to download and install the latest version of PeerView from its website [54]. At the time of the experiment, the latest version was 0.95, i.e. still a beta but close enough to a stable release (1.0) that I found it acceptable to conduct an experiment since I did not intend to add any major functionality nor expect to find any critical bugs[2]. The download process took approximately 10 minutes which was considerably longer than expected. The delay was caused by severe network lag and thus probably an accurate reflection of the fluctuations that users would experience under less controlled circumstances. Upon successful download and installation which took place without much difficulty, the participants began carrying out their tasks. One of the participants, the 28 year old man, had been given a task list which was slightly different from that given to the two others. It instructed him to create a group for the other two to join and to edit a shared document to determine whether changes were distributed once they had joined. Both he and the other two participants were later asked (by the task list) to contribute to the topic tree corresponding to the shared document he had added. The difference in task lists was necessary to strengthen the collaborative aspect of the experiment. By setting up a group and have all users join it, and then collaborate, albeit in a rudimentary fashion, on a shared document, the experiment came to encompass not only the single user dimensions of PeerView, but also its collaborative functionality.

The first task for all three, however, was to adjust the preferences so that their name and the correct server information was entered. They did this by modifying the settings in the preferences dialog box and then restarted the entire program to make the settings take effect. The last step was necessary because instant application[3] had not yet been implemented. Afterwards, the participant assigned the task of creating a group carried that out while the other two added a number of documents to their respective panoramas. They then moved on to carry out the other tasks on their task lists as appears from the material in Appendix B.

4.3 Discussion

The PeerView usability experiment was limited in scope by having only three participants and a duration of only two hours including the time taken to fill out the questionnaires. In spite of

[2]Program errors capable of triggering runtime failures such as system crashes.
[3]Often signaled by an "Apply" button in such dialog boxes.

this, it yielded a fair amount of useful information and gave me reason to believe that increasing the duration and number of participants would probably have given a quantitatively, not a qualitatively different amount of information. I therefore trust the results to be substantial enough to safely draw conclusions about the usability of PeerView in its current form. The use of several techniques — controlled experiment, user observation, questionnaire and informal discussion after the experiment — was chosen to lend the findings more credibility. That particular set of techniques was decided upon because I deemed it well suited for gathering a diverse set of data and because it comprises the most widely used forms of evaluation among groupware researchers. In [45], Pinelle analyses the evaluation performed in 32 of 45 articles on groupware (the remainder did not contain formal evaluation), and finds that laboratory experiment is the most commonly used evaluation type, and that user observation, interview and questionnaire techniques, in that order of prevalence, are the most popular ways of carrying out the evaluation.

The experiment produced information on both the *single user* and *collaborative* usability of PeerView, as appears from the transscript (Appendix B.2). After the user[4] assigned with the task of creating a group had done so, the other users joined the group and subsequently received copies of the artifacts that other users had placed in their panoramas. They were then each given the task of contributing to the topic tree associated with a specific document (referred to as "test_ gruppe.txt" in the transscript) added by the user who created the group (the only document he was asked to put in his panorama). Towards the end of the experiment, the same user modified "test_ gruppe.txt" and the other users checked their copies to see whether the change had been distributed correctly, i.e. whether their copies of that particular document had been updated to reflect the change. I had, of course, tested the update functionality separately, so the purpose was not so much to see whether it worked according to specifications, but rather to round off the collaborative component of the experiment.

4.3.1 Experimental findings

The experiment participants indicated or remarked repeatedly (e.g. transscript item 28,40,62 and questionnaires) that they had problems understanding the system dialogue or the consequences of the actions they performed. This was mostly due to the dialogue being in English, using unknown or counterintuitive terminology (such as referring to the nested topic structure in the bottom portion of the window as a "topic tree") or to the interface being less informative than could be desired (as in the case where one participant asked how he could tell whether he had joined a group [transscript, item 46]). The participants did not use the help system although they were encouraged to do so which was probably due to me being present while the experiment took place and answering questions on how to interpret system messages and other trivial issues such as which keys to press to produce a tilde character).

There were a few concrete interface problems. One was that highlighted text appeared at a different scale from the highlighting itself [transscript, item 67], which made it seem as if a smaller portion of text was highlighted than was actually the case. This confusing phenomenon was referred to as an optical illusion by the participant who noticed it, and his subsequent attempts to align highlighting and text by scaling the document to 1:1 magnification further revealed that panorama scaling was perhaps overly sensitive to user input since he

[4]He is referred to as "Paul" in the transscript.

had difficulty controlling it. An unrelated problem was discovered by the woman participant who experienced difficulty with the visible documents box in the right hand portion of the toolbar panel. Her monitor operated at a lower resolution than the other monitors and as a consequence the visible documents drop-down box could not expand downwards. In effect, it “dropped up” rather than down by expanding in the upwards direction. This rendered its contents difficult to access as indicated in [transscript, item 42]. None of the participants seemed to notice or have use for the draggable resize dividers that separate the topic tree from the message area in the bottom of the display and the panorama from the discussion forum. At one point [transscript, item 54], I tried to draw the participants' attention to the dividers to see whether they ignored them because they were unaware of the resize functionality or because they had no need for it. None of them used the dividers after I had made them aware of their purpose, but I doubt that can be taken as indication that they are unneccessary since different participants seemed to grapple a bit with fitting a document in the visible viewport or seeing the title of a topic in the topic tree. They might therefore have benefited from the resize functionality if they had been more comfortable with it. It would probably be desirable to somehow communicate more clearly that the dividers bars can be used for resizing, perhaps by annotating them with a label containing descriptive text.

A more abstract, general problem reported mainly in the comments section of the questionnaires was that the participants had difficulty understanding the context of their actions because it was unclear what audience PeerView was aimed at. This became clearer during the informal discussion afterwards when I explained the idea of making a prototype design before making a fully fledged program, but the participants nevertheless indicated that they felt somewhat bewildered during their work with PeerView. I suspect that this reaction is typical when users meet a program with a set of expectations shaped by work with commercial applications aimed at solving specific work problems. If development had gone through several iterations, each concluded by experimental evaluation, the problem could have been countered by using the same group of test participants each time since they would then be accustomed to the work process and its premises. The drawback of that approach is of course that such a group of seasoned test participants would quickly become unrepresentative of the user audience at large, thus probably degrading the value of any experimental findings derived from their activities.

The collaborative aspect of the experiment (as described in Section 4.2) was too limited for me to draw any firm conclusions, but it did seem to indicate that the participants had little awareness of each other. This applies to both group awareness (in the sense of being cognizant of the other participants) or workspace awareness (being aware of the others' actions in the workspace) as indicated by [transscript, item 48, 74] where users display signs of not feeling part of a collaborative unit, seemingly in part because the PeerView interface does not make it sufficiently clear that they are. When addressing me, the participants occasionally made references to what they had overheard me saying to other participants which could be taken as suggesting that they would benefit from more direct communication facilities such as the “chat” that was planned for, but unfortunately not included in the PeerView 1.0 release due to time restrictions.

4.4 User evaluation

After the experiment was concluded, the participants and myself sat down to fill out the questionnaires handed to them and to discuss the experiment. The questionnaires were worded in a way so that they could be understood without specialized knowledge to minimize the potential for misunderstandings. They were divided into three Sections:

- A set of Likert scale propositions regarding specific usability issues. A Likert scale can be used to have informants indicate on a scale ranging from complete agreement to complete disagreement the degree to which they agree with a proposition.
- A semantic differential schema where the participants where asked to indicate on a bipolar scale the degree to which they felt a set of descriptive phrases applied to PeerView.
- A free form comments Section where participants where asked to write any comments or remarks they might have about PeerView.

The overall assessment of PeerView was favourable, but the comments indicated some of the usability problems discussed earlier, namely that the terminology used by the PeerView interface in its messages and widgets was at times difficult to understand, not only because of the language being foreign to the participants who were all Danish, but also because they used terms that were unknown to them. Another problem was that participants lacked a clear statement of whom PeerView catered to. One participant wrote that: "I have the impression that you would quickly grow accustomed to the programme and grow to like it in the proper setting" [questionnaires]. But the subsequent discussion revealed that, like the others, he had difficulty pinpointing exactly what that setting should be. I suggested that additional functionality, such as that to be included in InSiter, might make it more useful to them and their work colleagues as well as other groups, and they seemed to agree that enhanced awareness in the form of video mosaics of coworkers (for example in the style of PortHoles [17]), chat facilities and different workspace contents at different magnifications (semantic zooming) were worth introducing.

Chapter 5

Implications of PeerView 1.0 for future systems

5.1 Introduction

The PeerView 1.0 prototype is exactly that: a prototype for new ideas whose purpose is to demonstrate their feasibility and viability. It is intended as both a test vehicle, a useful tool and a learning experience on which to base design of future systems. In this Section, I will analyse *a posteriori* the design, implementation and development of PeerView 1.0 to examine how a future rendering and review system (outlined in Chapter 6) can benefit from the findings of the PeerView project.

Of course, the below applies not only to successor systems, but also to PeerView 1.0 itself. The findings and conclusions might therefore conceivably be incorporated into the current version of PeerView to yield a significantly upgraded product which could aptly be called PeerView 2.0! However, I did not intend PeerView as an "open-ended" project and the experience from building version 1.0 is so comprehensive that I have found it most appropriate to use a different name, namely *InSiter*, for the successor project. PeerView development may continue, though, since the project has been made open source. Time will tell.

5.2 Design analysis

5.2.1 Introduction

The discussion in the following Sections is mostly general, concerned with the guidelines for design that can be derived from PeerView. Details are only brought in where appropriate to this discussion as the purpose is to facilitate future development, not to formulate specific improvements to the current PeerView design.

5.2.2 Interface

In retrospect, it is clear that the design of the PeerView GUI rested on implicit assumptions about the suitability of metaphors as design devices. That is, by readily adopting the desktop metaphor (for the scalable desktop) and a debating society metaphor (for the discussion forum)[1], I accepted the commonly held belief among GUI designers that design based on metaphors equals user friendly design [9, p. 5]. This assumption is not necessarily justified, as Blackwell documents in his Ph.D thesis. Based on comprehensive experiments where novice and expert users were asked to solve various exercises using metaphorical and non-metaphorical diagrams and notations, he concludes that [9, p. 164]:

> "Most of the experiments described in this thesis ... have tested the assumption that diagrams are universally beneficial in problem solving and design. This is obviously not the case — even within the scope of these studies, a simple distinction between 'experts' and 'novices' has revealed not universal benefits, but a large difference between individuals with different amounts of experience."

How, then, can one account for the inarguable success of the desktop metaphor as popularized by the Apple Macintosh and Microsoft Windows operating systems? Blackwell does not attribute this success to the metaphor as such, but argues instead that [9, p. 160]:

> "The demonstrable advantages of graphical user interfaces, as with many types of diagram, can be explained in other terms. Direct manipulation — representing abstract computational entities as graphical objects that have a constant location on the screen until the user chooses to move them — facilitates reasoning about the interface by reducing the number of possible consequences of an action ..."

This seems to validate the choice of a desktop GUI design, although not by virtue of its metaphor design philosophy. I believe a similar argument applies to the discussion forum design as it relies on a form of direct manipulation that is both familiar to many users (for example from the graphical directory browsers found in many operating systems) and fits the description given by Blackwell. This seems to be corroborated by the predominantly positive assessment made in the questionnaires that the evaluation experiments completed (see Appendix B).

A useful lesson to be derived from the above is that one cannot ensure a sound design by relying exclusively on textbook recommendations and established dogma in the design community. Instead, the interface decided upon during the initial design phase could be translated into a working prototype or a mock-up, i.e. a "scaffolding" with no real functionality, which could then be subjected to usability testing on a group of end users. The empirical data thus gathered could then be used to modify, if necessary, the interface design, and the test cycle should then continue until the GUI design is deemed satisfactory. Of course, the initial design would rely on a set of implicit assumptions exactly as the PeerView design did. This seems inevitable since end users cannot, in general, be expected to describe a priori and in detail their ideal

[1]I have no direct evidence to support that the original USENET design of a hierarchically organized newsgroup was inspired by debating societies or similar discursive fora, but my choice of that design was influenced by such considerations.

interface although most can, presumably, give constructive feedback on a concrete proposal presented to them. The designer should therefore ensure that his assumptions are grounded in empirical fact and well reasoned rationale, not only in commonly accepted practice and beliefs.

5.2.3 Functionality

It is clear that the actions that users can perform using PeerView constitute a minimum of functionality for a usable application. This is hardly surprising, considering that it is a prototype, but it does indicate that any successor system should not restrict itself to the ideas now explored if the scope of the research is to be increased. It should provide a fully fledged system that has the width and depth necessary to be useful in realistic work settings with large groups and large masses of data. The objective should not necessarily be to compete with similar, commercial systems, but to elaborate on the ideas embodied in PeerView, *and* demonstrate that they can "survive in the wild".

The bulk of user interaction in PeerView 1.0 is with the scalable desktop, i.e. the panorama on which artifacts are placed. The issues raised by adopting a desktop metaphor design were discussed in Section 5.2.2, and the rationale behind using a zoomable interface was presented in Section 3.1. The actions that users can perform on the panorama, i.e. its functionality, was limited to the essential operations as discussed in Section 3.1. Hence, there is ample room for improvement, both by extending the set of operations, although this would be merely a quantitative improvement, and by using a more sophisticated zoom mechanism which could make the entire interface qualitatively different. One such enhancement is *semantic zooming*, where the level of elevation (i.e. distance from the origo of the z axis) determines the representation used in the zoomable display. The reverse — that the choice of representation determines the elevation — has also been tried and is called *goal-directed zoom* in [59], where the authors describe a system that allows users to select an object and choose from a set of representations how it should be rendered. The system itself determines the appropriate form of representation at a given elevation by applying what the authors refer to as the Principle of Constant Information Density, which states that "the amount of visual information per display unit should remain constant as the user pans and zooms" [59, p. 306].

The Jazz library can be extended to support semantic zooming [8], so implementing such functionality is quite feasible using known technologies. I believe it could be of particular use in systems where the rendering was extended to arbitrarily large sets of artifacts and the process by which they were brought about. The reason is that it would allow a series of mutually supporting representations of group work to be rendered within the same display space and navigated in a convenient and familiar fashion. One example might be to have a project represented by a diagram at high elevations, by clusters of developers represented by photographs and brief biographical synopses at medium elevations and by grid arrangements of artifacts at low elevations, as done in PeerView. Chapter 6 elaborates further on this point.

Future rendering systems could of course abandon the idea of zoomable interfaces altogether and use entirely different modes of presentation. I believe, however, that ZUIs that support semantic zooming are a near ideal "$2\frac{1}{2}$D" alternative to the conventional 2D and 3D displays, both of which have inherent limitations, as discussed in Section 3.1. Consequently, future systems would benefit from extending the PeerView 1.0 interface both qualitatively, by introducing elevation dependent representation through semantic zooming, and, possibly, quantitatively by giving users a wider selection of options than that listed in the PeerView 1.0 artifact pop-up menu.

In terms of support for collaboration, PeerView 1.0 gives users simple *workspace awareness*, i.e. who the other members are and what they are doing (to artifacts) in the shared workspace, and has very rudimentary or no support for other forms of awareness, such as *group-structural awareness* (awareness of people's roles in a group, their status and positions on issues) and *social awareness* (awareness of people's emotional and intellectual states) [47, p. 2]. In [26], Gutwin and Greenberg find a strong positive correlation between the quality of support for workspace awareness in groupware applications and the performance and satisfaction of participants in experiments based on those applications. Their subjects used a *radar view* window which is referred as a "miniature" [2] to get a global perspective on the workspace and to follow the activities of other users. PeerView 1.0 naturally supports the former of these activities, but not the latter and that issue should be addressed by any successor systems. In fact, Gutwin and Greenberg write that [26, p. 517]:

> " ...the main finding of the study is that adding workspace awareness information to the miniature — visual indications of viewport location, cursor movement, and object movement — can significantly improve speed, efficiency, and satisfaction. These awareness components should be included in shared-workspace applications."

This point seems to further underscore the qualities of a zoomable interface as compared to 2D and 3D alternatives which typically would require some dedicated miniature to be grafted onto the main interface to obtain the effect recommended by Gutwin and Greenberg. Zoomable interfaces allow an overview effect to be completely integrated in the ordinary operation of the interface, as PeerView demonstrates. Future systems could elaborate on that effect by integrating the features discussed in the above quote without subtracting from the interface's ease of use, i.e. without inconveniencing the user.

5.3 Implementation analysis

5.3.1 Introduction

In this Section, I discuss those aspects of the PeerView implementation that carry useful lessons for implementation of InSiter and of system implementation in general. As with design, those lessons are mostly general and not specific to implementation details, but the discussion does bring in concrete examples where necessary.

5.3.2 Implementation language and technologies

As discussed in Section 3.1, the implementation of PeerView was facilitated by using readily available component technologies to shorten development time. Jazz [34] and JSDT [?] were the most visible such component packages, but also the Swing interface classes that are usually

[2] A scaled down representation of the workspace proper.

thought of and used as an integral part of the Java Development Kit were a component technology that helped make implementation by a single developer possible. I experienced little difficulty using these component technologies and ascribe this in large part to the fact that all were written in pure Java, i.e. without use of code specific to a particular platform or architecture, so called native code. Also, Jazz, JSDT and Swing all have well designed APIs that make interfacing them with one's own code quite easy.

Judging from this experience, component-based development seems to be good practice but the following should be borne in mind:

- The interface of components should be well defined and well behaved, meaning that the API documentation should provide descriptions of both the syntax (how to use) and the semantics (what happens when used) of each component method along with a description of its side effects and complexity. If these are not available, the developer may realise that the savings in development time was too dearly bought with deficiencies in the end product. As Meyer notes in [41]: " ...the quality of a component-based application is defined by the quality of its worst component."

- The components libraries used should be compatible, preferably by being written in the implementation language of the application proper and by adhering to the same design principles. The problems that can be incurred by using code from separate languages are well known: one poignant example is the need to distinguish between different calling conventions for functions in `C++` code where for example one must specify that a function should be called with the `C` language convention by prefixing the function definition with `extern "C"`. The problems that may arise from differences in design may be more subtle, but no less real. One example could be to make use of two component libraries, one of which was highly optimized for some time critical operation such as real-time rendering, the other designed with security concerns as the primary concern and performance as a distant second or third. Such grinding incongruency between design goals could result in components cancelling out each other's respective benefits, making for a whole that is actually less than the sum of its parts.

- Component based development is not merely an question of picking off-the-shelf components, connecting them in some appropriate way and then executing the resultant mélange. The developer must negotiate between the ideal implementation of her design and the trade-offs involved in using component technology. In PeerView's case, for example, the Jazz library was used to implement the zoomable interface, i.e. the scalable desktop, but there were alternatives that had a differently balanced design. OpenGL [44] was one such alternative, as indicated in Section 3.1, which has a much broader range of graphics operations than Jazz, but no specialized suppport for the 2D zooming operations needed to scale text and bitmap graphics at interactive rates on even low-end systems [7]. One could envision a more sophisticated system that used a combination of zooming and 3D navigation (for example by means of a scale space representation [20, p. 110]), and might therefore have to rely on a library such as OpenGL, which in turn would mean that the minimum system requirements for end users would have to be raised significantly.

Component-based development therefore affects the entire development cycle, not just class design and implementation. Trade-offs have to be made throughout to find the best balance between the envisioned end product and the degrees of freedom afforded by available component libraries.

5.3.3 Representation of properties and messages

An application like PeerView typically contains a set of mutable and constant values that define its behaviour and interface. An example of the former are error messages most of which are typically immutable and triggered by specific (exceptional) conditions in the application. An example of the latter are *preferences* which determine the general appearance and functionality of an application and can usually be defined by the user.

It is a basic convention in good programming style not to use so called "magic numbers" [40, find side], i.e. constants that appear by their literal value in the code and not a symbolic name. This applies not only to numerical constants, of course, but also to string literals such as user messages. Consequently, all such literals are represented by symbolic names in PeerView. The literals themselves are stored in designated classes called `ServerConstants` and `ClientConstants` and are accessed through access methods, not directly, so a literal whose symbolic name was `windowSize` would be accessed through a function call to a method named `getWindowSize`. This makes it possible to concatenate the constant with one or more arguments to the `get` method before the resultant value is returned to the user. One example is:

```
public final static String getREMOVING_DISTRIBUTED_DOCUMENT(String documentName,
String senderName)
{
return senderName + " has requested that " + documentName + " be removed from the
panorama.";
}
```

Since some constants are shared by the client and server application, the `ClientConstants` and `ServerConstants` are derived from the class `SharedConstants` which contains those shared constants and the associated access methods. Both the server and client constants class contain a set of default preferences which are constants used when the user has not specified any value for a specific property.

This design is not new; using classes of static constants accessed through methods is well-known practice in Java programming, but by applying this technique consistently as in PeerView, the textual component of the user interface is separated from the implementation. This means that if a different version must be installed, either to correct an error (a "bug fix") or to change the language, it can be done without requiring a new full install by the end user, or indeed a compilation by the developer. Users can instead download a small patch to the finished product and install that as instructed. This is a simpler method than the internationalization scheme described in [35] and could therefore be useful when the implementation should adhere to good style conventions, but not be bogged down in the details of internationalization.

5.3.4 Implementation of the distributed architecture

The distribution layer or data sharing mechanism of PeerView is implemented using the Java Shared Data Toolkit (JSDT) as discussed earlier. The JSDT provides a high level interface for use by developers of collaborative applications (among others) which is independent of the underlying communication protocol. That is, calls to the JSDT API do not specify or presume a particular protocol[3] and therefore do not need to be changed if one protocol is replaced by another. As a result, one can use the JSDT with any protocol that allows data to be exchanged among a number of collaborating entities. In its current distribution (version 2.0), the JSDT supports a TCP or UDP sockets protocol which in theory could be used for both LAN, WAN and Internet traffic. However, in practice, many firewalls will block attempts to make a direct TCP connection between two machines and so the JSDT also supports an HTTP protocol which can "tunnel" through such firewalls provided the server side is properly configured. This is why PeerView allows the user to specify either protocol. The user must also specify the port to use as target when packets are routed through the network, and she must make sure that the firewall on both the server and client side allow traffic to pass through this port.

Sockets based traffic is a simple form of data exchange and for collaborative applications, IP multicast is a potentially better solution. Whereas unicasting data as done in TCP communication requires the sender to send separate packages to each receiver, a multicast protocol allows the sender to send just one package and then have it replicated by multicast routers on the network and forwarded to the intended recipients, possibly through several more replications. PeerView does not support multicasting in its present form because it is primarily intended for work environments where the data density, i.e. the amount of traffic generated, is moderate and not sufficient to "swamp" the network. However, an application that must scale to more demanding environments will probably need to use multicasting protocols as data traffic increases to achieve acceptable performance on all but dedicated networks.

As mentioned, JSDT can be made to work with any suitable protocol implementation, according to its documentation, but for multicast there is fortunately a way to minimize time consuming experimentation. It is, once again according to JSDT documentation, possible to couple the JSDT with the LRMP (Light weight Reliable Multicast Protocol) package from Inria [39] with no great difficulty. The LRMP implements a true push service meaning that clients need not be aware of the server's name nor have a port number to connect to. Instead, they subscribe to an information channel and receive data when it is broadcast, so the distribution scheme resembles a radio channel rather than an telephone line as is the case with the sockets channels. In addition, the LRMP architecture is based on IP multicast with its associated advantages of supporting both one-to-one and one-to-many communication as well as making effective use of available bandwidth.

The primary reason PeerView was not implemented using the LRMP protocol was my reticence towards using a technology that I had no implementation ("hands-on") experience with. Sockets based communication is the default protocol in JSDT and consequently the best documented. Also, I was familiar with the basics of sockets programming and so felt more comfortably using that technology, knowing there were deadlines to be met for the PeerView project. In retrospect, I am not sure this was the right decision because the advantages of the

[3]The protocol may be configured via the JSDT API, but that does not detract from the orthogonality of the API as a whole.

LRMP (greater scalability, simpler user interface, more functionality) may have outweighed the cost in increased development time. It is, of course, difficult to assess since I did not experiment with the LRMP protocol and therefore have no experience to base an evaluation on. However, for any future project, InSiter in particular, a multicast implementation would be preferable to a sockets based implementation because the advantages of the former become so compelling as the scope of the application increases that it would be indefensible to use a simple unicast solution.

PeerView uses only a subset of the functionality in the JSDT library, namely the data distribution mechanism. The library also has support for data sharing which allows a group of clients to share a data reservoir (implemented as a byte array) and for implementing a security layer which enables controlled access to shared data and data distribution. This limited use of available functionality is not merely due to the limited scope of the PeerView application. I suspect that most collaborative applications, InSiter included, will have a greater need for a scalable, efficient, reliable data distribution layer than for auxiliary features such as data sharing, security and the like. The reason is that with a sound data distribution layer such as LRMP multicasting in place it is relatively straightforward to implement features such as security (for example by maintaining a registry of clients and their access privileges) and data sharing (by using a byte array and a simple token based access scheme) on top of that layer. If this is the case, then much of the JSDT functionality is unnecessary for the implementation of InSiter, although the JSDT is far simpler than the alternatives surveyed and mentioned in Section 3.1. Instead, the best solution may be a library aimed purely at data distribution and therefore optimized for that purpose. The Web Canal [39] tool from the French research institute Inria seems a good choice because it uses the LRMP protocol also from Inria and therefore has the desirable properties of that protocol as discussed earlier. Furthermore, Web Canal is freely available and pure Java so neither licensing nor compatibility concerns need hinder or limit integration into one's own applications.

The experience gained from implementing the PeerView distribution layer is perhaps the most valuable implementation lesson to be derived from the entire project because the choice of data communication library has such a significant impact on both the internals of the implementation and the end user experience (by influencing network load, responsiveness and ease-of-use).

Chapter 6

InSiter

6.1 Introduction

In this Chapter, I present a tentative design of a system for rendering and review that builds on the experience garnered from the development and evaluation of PeerView. This system is considerably more ambitious than PeerView and is intended for practical, day-to-day use in a realistic setting, be it in academia or industry. The working title of this system is *InSiter*, a name chosen to signal the organization-wide scope of the rendering it presents to its users. In the following Sections, I describe the intentions and design philosophy of InSiter and present the interface design of its main components, and discuss in general terms how this could be implemented.

6.2 User audience

The intended users of InSiter are the same as for PeerView, namely first and foremost software developers, but as with PeerView, InSiter should be designed and implemented so that is accessible to most information workers (from office clerks to civil engineers) without extensive training. I believe this is not only feasible, but also desirable. Not just because it widens the potential user audience, but also because it forces design work to focus on the essentials of the product and how they are best communicated to all users. In Section 3.2.1, I described the principles of simplicity in connection with interface design and how they are propounded by industry and academia pundits. This credo of the simple and rejection of the overly elaborate[1] recognizes the inherent limitations of the human cognitive system and, arguably, has some intrinsic aesthetic appeal (see also [55]). The InSiter design should therefore cater to its intended audience, i.e. software developers, but at the same time be as inclusive as possible.

6.3 Design objectives

PeerView was intended as a prototype to test the viability of the ideas of rendering and review, i.e. both the usefulness of a design philosophy based on those concepts and the feasibility of

[1] Facetiously but incisively referred to as the "keep it simple, stupid" or KISS principle by some.

translating them into reality. It did yield useful information on the latter aspect, but little on the first since it was, by necessity, limited in scope and therefore did not lend itself to the type of prolonged experimentation in realistic settings needed to make conclusive statements on usefulness.

InSiter should not be an experimental system — it is intended for “production use”, but it might still produce valuable experimental results by allowing the actual usefulness of rendering and review to come before a day, so to speak, by being used by real groups working on real projects in real organizations. The purpose is therefore to produce a workable (in the sense of being accessible and useful) and working (in the sense of being well designed and implemented) rendering and review system that builds on the experience from PeerView to provide users with a common information space and facilities for reviewing and discussing its contents.

The InSiter interface will be a ZUI (zoomable user interface) much in the same way as the PeerView interface, but it will employ a wider range of ZUI techniques because it must allow users to insert more types of artifact into its interface and allow them to apply a greater and more diverse set of operations to those artifacts than PeerView did. This is necessary to give users the freedom to customize the information space to suit their needs.

The PeerView interface was *document-centric* in that the only type of artifact that could be placed on its scalable desktop was documents, albeit their contents ranged from bitmaps to HTML code. The InSiter interface retains this emphasis on documents since such finite strings of symbols are after all the only artifact that information workers can produce, but it widens the scope to include the relationships between those artifacts, the people who create them and the organizational relationships between those people. By doing so, InSiter aims to render not only the outcome of a project but also the context it which it is brought about.

InSiter will allow users to form collaborative units (which will be referred to as groups for the time being) and will give them more extensive communication and review facilities than PeerView did. It will support a live video feed of each group member to be placed in the panorama so as to provide a video mosaic similar to that discussed in [17]. Further, InSiter will allow members to communicate directly on a member-to-member (unicasting) or member-to-group (broadcasting) basis by exchanging text messages which can contain hyperlinks to artifacts in the common information space and thus used as a simple means of initiating collaboration on a specific artifact. InSiter will give users the opportunity to discuss individual document artifacts by providing review facilities similar to what is found in PeerView. There seems to be little need to introduce new review facilities since the discussion forum concept in PeerView captures most of what transspires in a simple code review session which is what inspired it. Usability testings of early version of InSiter can probably reveal whether more sophisticated review facilities are desirable and so I leave those considerations to a later, more advanced stage of design.

InSiter will generate and store statistical information on document artifacts to facilitate analysis of them.

6.4 Interface design

6.4.1 Introduction

One of the most difficult design decisions is where to begin. After all, there is nothing to hinder design from beginning with how to ship the product to end users before describing how architecture and interface should be designed. However, the interface is a good place to start because it constitutes a concrete, visible commitment that can be understood by both end users and developers, and therefore sets a lower threshold for what subsequent layers of functionality and architecture must support.

Figures 6.1 to 6.6 are mock-up screen designs showing how the InSiter panorama will appear to the user. This is a simplified representation that captures the overall intent, meaning that details may differ in the final design. Based on the experience from PeerView and the experimental evaluation described in Chapter 4, I believe the exact nature of those details could favourably be decided on through repeated, i.e. iterative, evaluation of progressively refined design proposals. That way, the final design would likely be an accurate reflection of user needs, although the rules of thumb applied in the design of the PeerView interface (see section 3.2.1) can also help guide interface design.

However, for any evaluation to take place, there has to be some interface in place to begin with, and the below screen designs serve that purpose. Section 6.4.3 then describes the rationale behind these designs.

6.4.2 Screen designs

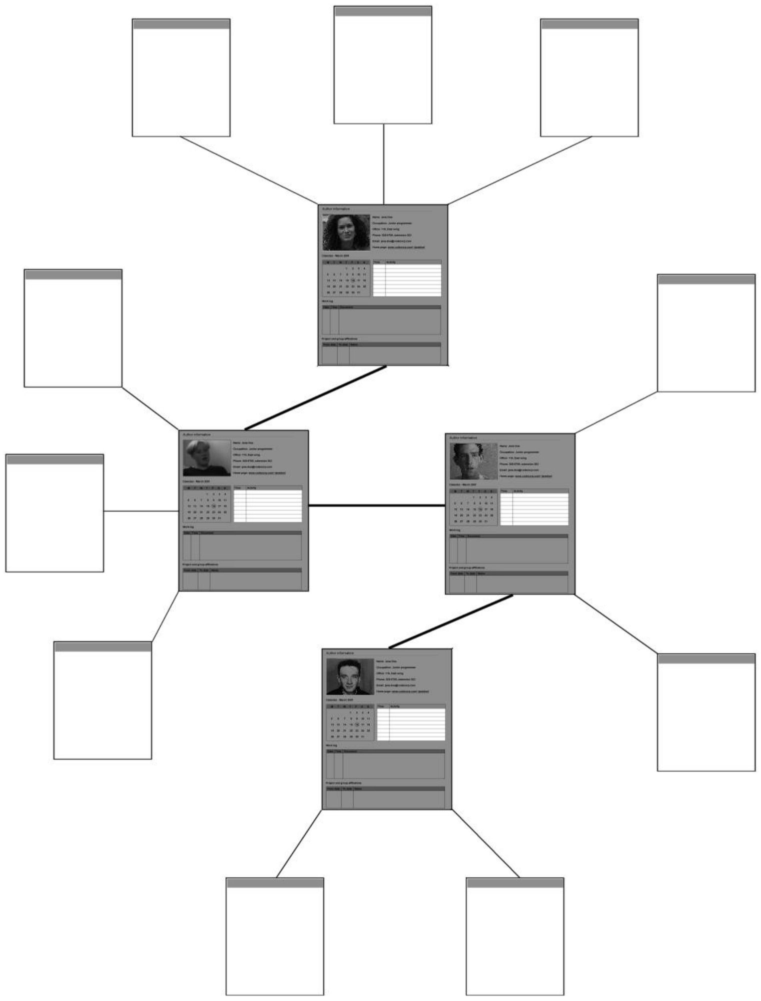


Figure 6.1: A mock-up of an InSiter panorama. Four author nodes are connected to a number of document nodes, all of which can be translated and scaled individually by the user. A variety of layout managers can be used to automatically lay out nodes so that they are arranged by author, by modification, by some geometric arrangement (as with the grid layout manager in PeerView), or by some other criterion entirely. The user can zoom manually or by double-clicking any node or edge.

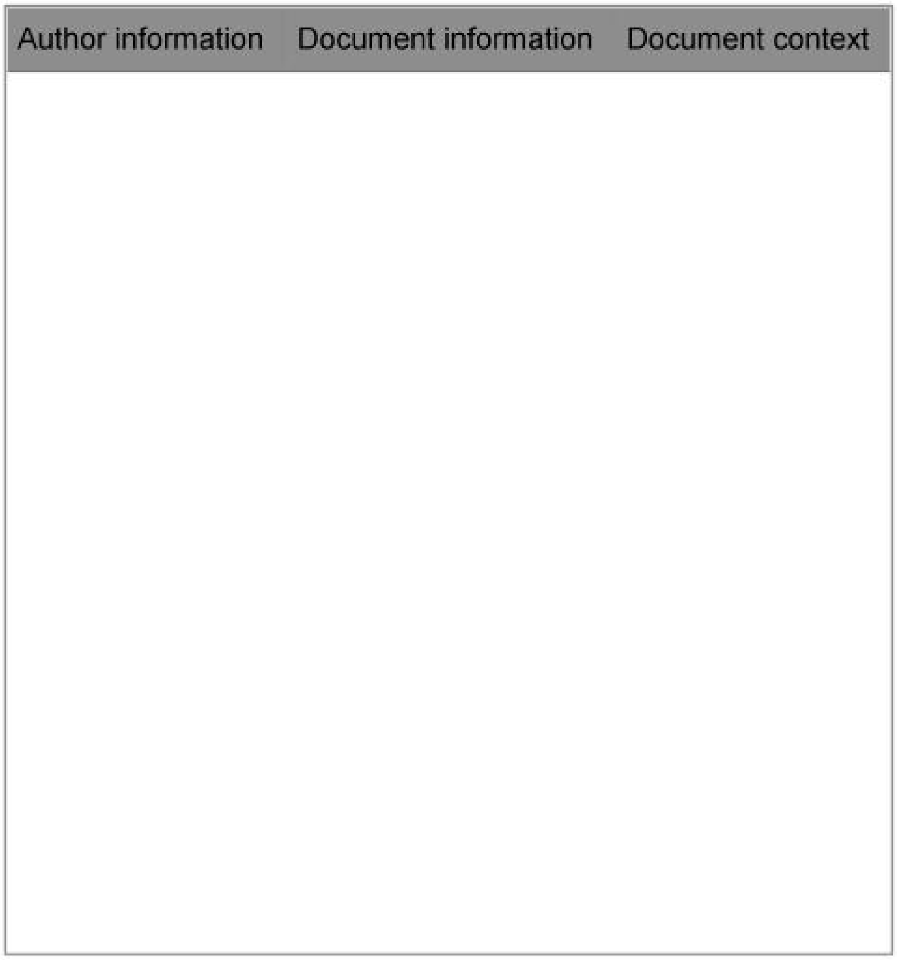
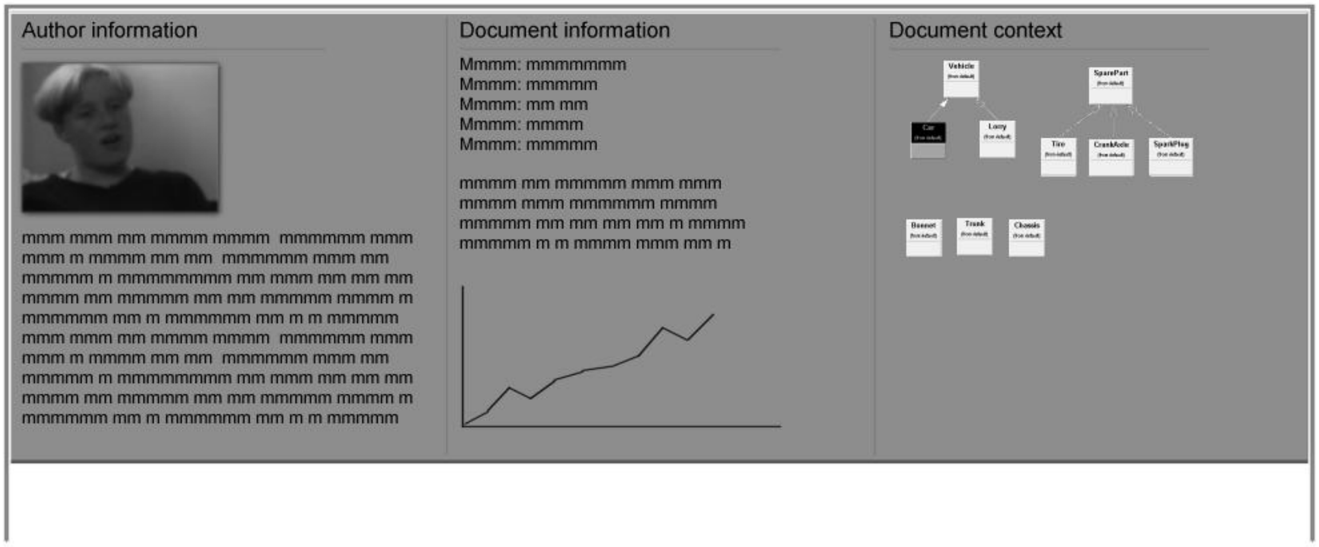


Figure 6.2: Centered view of InSiter document. The user can double-click each of the three text items in the header bar to zoom quickly to a centered view of either one.

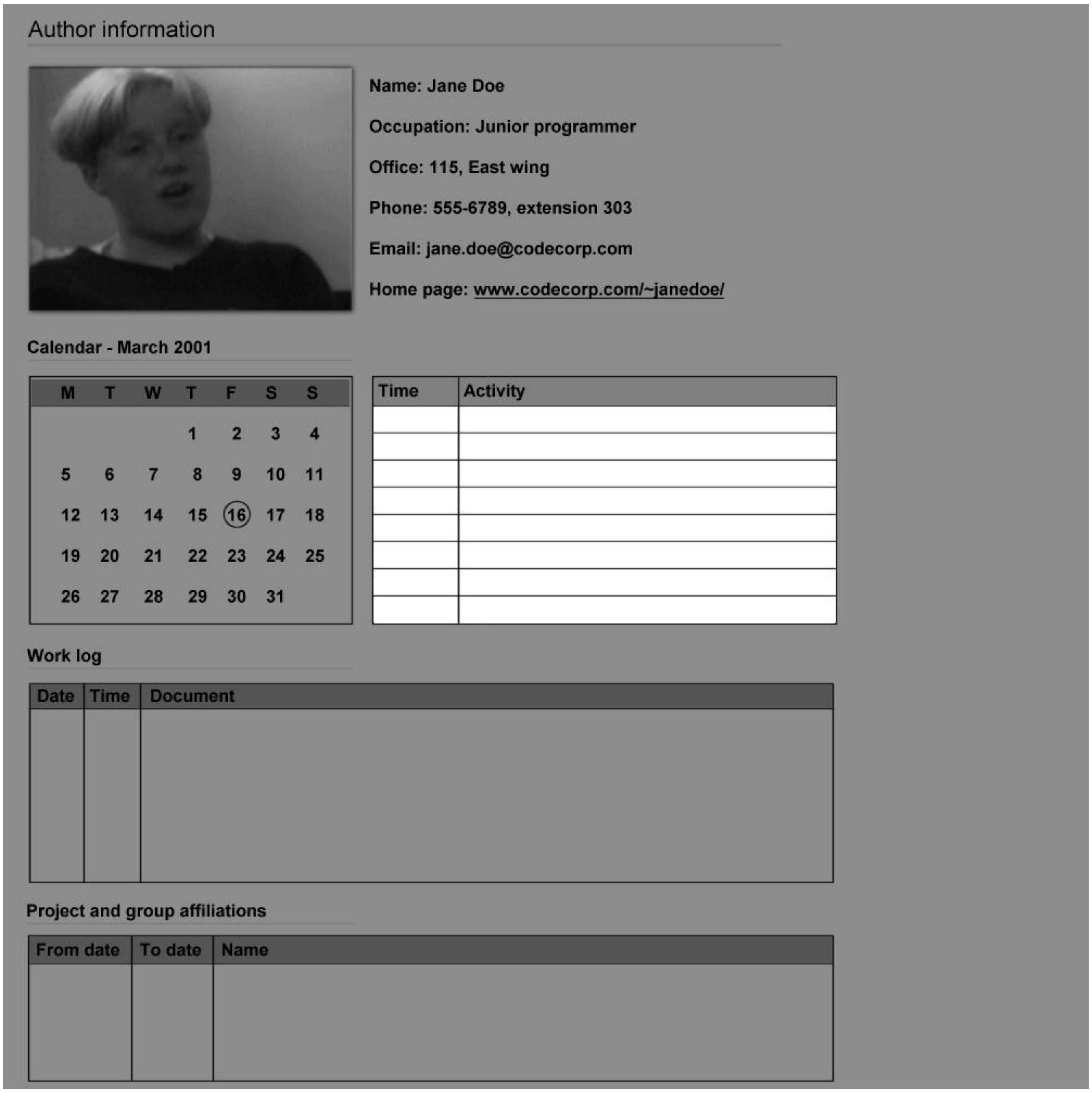


mmm mmm mm mmmm mmmm mmmmmm mmm
mmm m mmmm mm mm mmmmmm mmm mm
mmmmm m mmmmmmmm mm mmm mm mm mm
mmmm mm mmmmm mm mm mmmmm mmmm m
mmmmmm mm m mmmmmm mm m m mmmmm
mmm mmm mm mmmm mmmm mmmmmm mmm
mmm m mmmm mm mm mmmmmm mmm mm
mmmmm m mmmmmmmm mm mmm mm mm mm
mmmm mm mmmmm mm mm mmmmm mmmm m
mmmmmm mm m mmmmmm mm m m mmmmm

Mmmm: mmmmmmm
Mmmm: mmmmm
Mmmm: mm mm
Mmmm: mmmm
Mmmm: mmmmm

mmmm mm mmmmm mmm mmm
mmmm mmm mmmmmm mmmm
mmmmm mm mm mm mm m mmmm
mmmmm m m mmmm mmm mm m

Figure 6.3: Close-up view of InSiter document. The user has manually zoomed to a magnification that allows her to see all three portions of the document header bar. From here, she can double-click any of them to have it centered automatically or continue the manual navigation.



Author information

Name: Jane Doe

Occupation: Junior programmer

Office: 115, East wing

Phone: 555-6789, extension 303

Email: jane.doe@codecorp.com

Home page: www.codecorp.com/~janedoe/

Calendar - March 2001

M	T	W	T	F	S	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Time	Activity

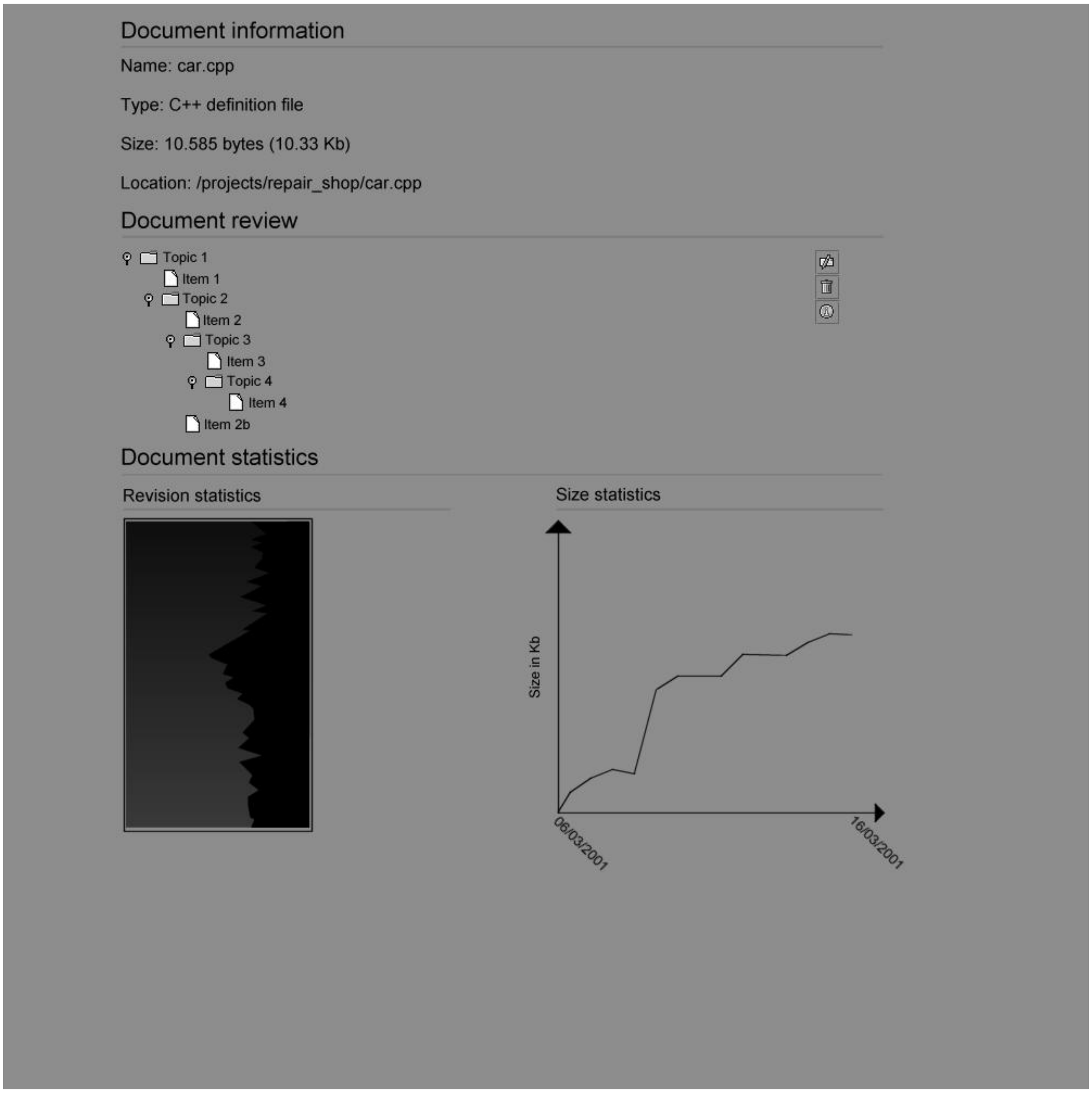
Work log

Date	Time	Document

Project and group affiliations

From date	To date	Name

Figure 6.4: Author information page. This portion of the InSiter document header bar contains information on the author of the document and a live video feed of her (if available).



Document information

Name: car.cpp

Type: C++ definition file

Size: 10.585 bytes (10.33 Kb)

Location: /projects/repair_shop/car.cpp

Document review

- Topic 1
  - Item 1
  - Topic 2
    - Item 2
    - Topic 3
      - Item 3
      - Topic 4
        - Item 4
    - Item 2b

Document statistics

Revision statistics

Size statistics

Figure 6.5: Document information page. This portion of the InSiter document header bar contains summary information on the document and a revision diagram (lower left) that shows an outline of the document along with an indication of when each line was last modified (colour coded with red indicating recently modified and blue the opposite).

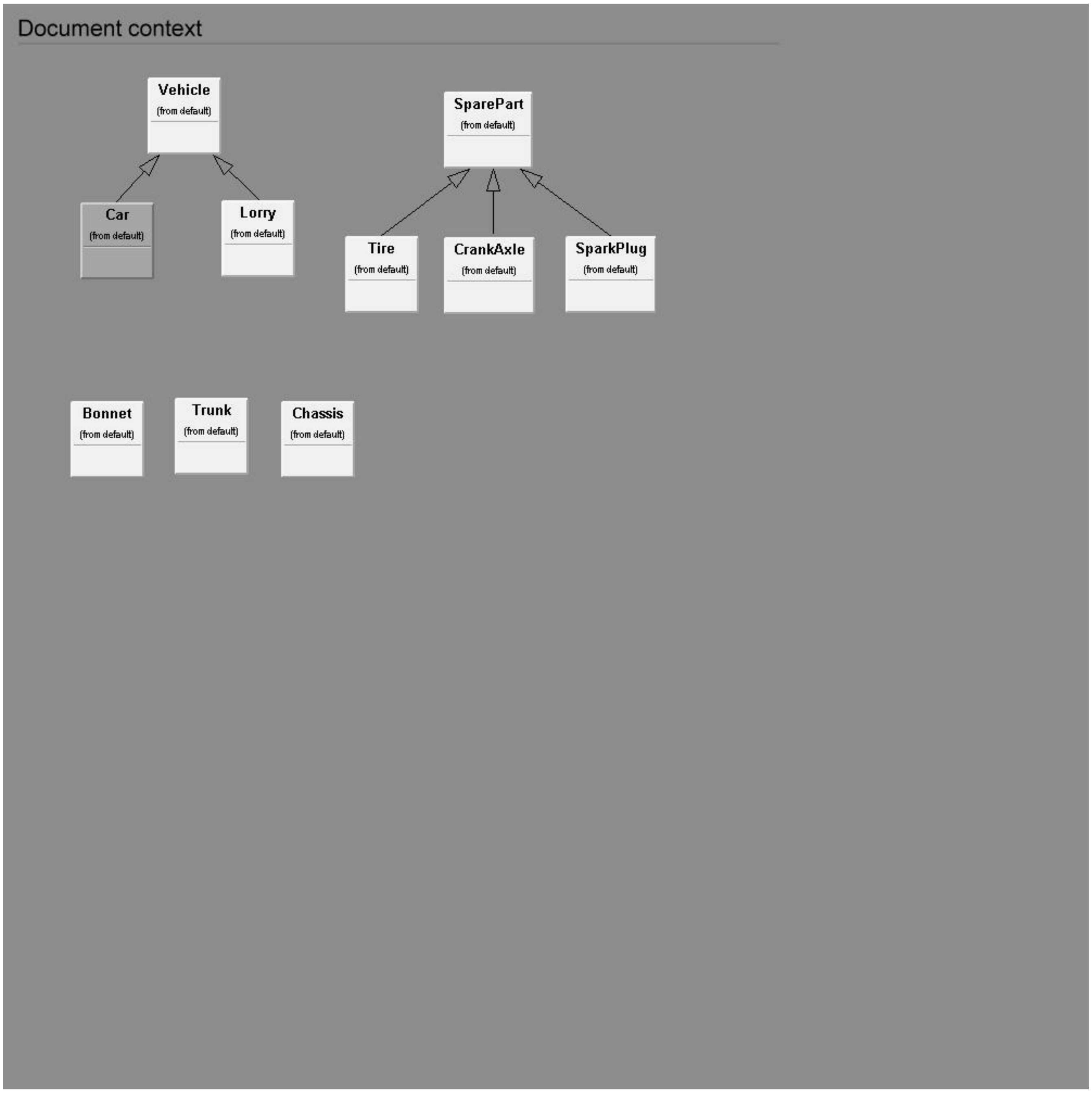


Figure 6.6: Document context page. This portion of the InSiter document header bar contains a user customizable representation of how the document relates to other artifacts. In this case, the document in question is a code file and its context is therefore appropriately represented by a UML class diagram.

6.4.3 Design rationale

InSiter aims to provide users with a comprehensive common information space as is reflected by the above screen designs. Section 6.5 describes the overall architecture that must support the proposed designs, but the rationale behind the inclusion of such features as video feeds and code statistics merits separate attention.

Video broadcast

In [17], Lee et al. describe their experience with a system called Portholes "that allows distributed work groups to access information related to general and peripheral awareness" [17, p. 385] and report a mixed user reaction. However, they also find that user reticence was attributable to such factors as camera shyness, threat of surveillance, loss of privacy and lack of control, all of which they address with varying degrees of success by allowing users to edit, blur and remove frames from the video feed streaming from their camera. They also found that users request a "lookback" facility for determining who is accessing their video feed or has done so recently. In other words, users want to be aware of who is looking at them so that they do not feel passively monitored, but rather on par with whomever is observing them. User control and support for customization is therefore vital if the InSiter video broadcast system is to be successful. Portholes provides this through a preferences section for each user where he or she can regulate who has access to which images and how those images appear. I plan to incorporate a similar facility in InSiter, but the specifics of the design will depend on the type and quality of video used. However, users should in any case be able to edit out specific frames, i.e. block the image stream at will and at any point, and should be able to see at a glance who is currently looking at them (this reciprocal relationship between observer and the observed is sometimes referred to as "reciprocity").

As for the technical feasibility of implementing a video broadcast system, consider the following back-of-the-envelope calculations:

Notation:

f the size in bytes of each broadcast frame.

r the frame rate measured in number of frames broadcast per second.

c $\frac{1}{compressionfactor}$.

b the bandwidth in bytes pr. second necessary to broadcast $f * r * c$ bytes.

Assume:

- A resolution of 320 by 200 pixels and a pixel depth of 8 bits (sufficient for 256 colours or grey scale). Then $f = 320 * 200 * 1 = 64.000$.
- A reasonable frame rate of $\frac{1}{3}$ the real-time frame rate. Then $r = 8$.
- A 1:10 compression ratio which is a conservative estimate, judging from [29]. Then $c = \frac{1}{10}$.

Then:
$b = 64.000 * 8 * \frac{1}{10} = 51.200$ bytes $= 50$KB

Notice that this estimate is independent of the number of users in a group if IP multicast technology is used, as discussed in Section 6.5.2, since multicast allows the sender to broadcast to multiple recipients without multiple transmissions by instead making a single transmission to a group address which is then automically forwarded to the members of the group in question. 50KB is a fraction of the bandwidth available in standard local area networks and within the range of many private subscribers with the increasing availability of ISDN and ADSL services. InSiter will need to distribute other forms of data, but video will probably account for a significant if not dominant portion of the total bandwidth consumption and so the above is also an estimate of the magnitude of the total InSiter bandwidth consumption.

Code statistics

In [52, p. 315 ff.], Eick describes the *SeeSoft* system which translates large amounts of code into a visual representation consisting of long vertical columns corresponding to the files containing the code in question. Each column can show the contours of a portion of code, i.e. how it would appear if viewed at a distance or, equivalently and more pertinent to InSiter, from a high elevation (that is, zoomed out so there is significant difference between the z coordinate of the viewport and that of the code). Each pixel line in a column then corresponded to a line of code and was colour coded to indicate its most recent date of revision. A similar arrangement is used in Figure 6.5 where an outline of the entire document is shown in the lower left corner. Each line is coloured so as to indicate when it was last changed and the user can double-click any portion of the outline which will result in InSiter zooming out to an appropriate viewing distance and scrolling to the portion of text corresponding to the clicked area. This feature will most likely only be enabled for documents containing computer source code since its usefulness lies in the fact that formatted source code has a distinctive signature outline that enables experienced programmers to quickly identify relevant segments of code by inspecting that outline [52, p. 318]. Also, to gather the necessary revision statistics, InSiter must either interface with the development tools used to create the documents in question, or make a line-by-line comparison of successive file versions each time a document is updated. The latter is a better solution since it does not depend on any interfacing with the constantly changing and largely proprietary development tools in use today. It does, however, carry with it a processing cost and should therefore only be done where it yields a useful outcome. The information thus gathered can also be used to implement a special inspection mode that temporarily modifies the apperance of document artifacts to give a SeeSoft-like overview representation. This inspection would expand each document so that all of its contents was visible without scrolling and then arrange the documents beside each other so that they made up a linear range of columns. By scaling to an appropriate level and perhaps activating a optional colour coding to indicate revision history, the user could then use the resulting arrangement of artifacts in the same manner as SeeSoft.

Again, computer source code files are the type of document for which such an overview representation would make most sense because of the relationship between code formatting and semantics (if-then statements and switch-case clauses in the *C* language are usually indented in an easily identifiable fashion, for example). However, if the line-by-line comparison scheme suggested above was used, it would be possible to maintain revision statistics for other text document types as well. Future design refinements will have to address this question in detail.

The other type of statistical data envisaged in this design is the size evolution graph in the lower right portion of Figure 6.5. This is included because it gives a useful, if incomplete sense of how the document has evolved over time while at the same time indicating when it was created and what its current size without taking up significantly more space than those vital data alone would have taken. Also, it is trivial to implement since the size of each document will be monitored anyway in the process of maintaining updated copies of the artifacts placed in the InSiter panorama.

Document context information

In Figure 6.6, the example document used is shown in context using a UML diagram. This is an illustration of the general principle that when users add documents to the InSiter panorama, they can specify the relationships of those documents to other documents. They should be able to do this either by attaching a separate document containg the necessary information (such as a UML diagram in some portable format) or by entering the relationship textually or using a mouse. InSiter can then use this specification to render a document context pane as show in figure 6.6. This need not be a UML diagram, but can be any representation that can be rendered in the InSiter panorama such as a dependency graph connected to a document showing which other documents must be updated when that document is. InSiter could then be configured to automatically notify affected parties when a document is changed that other documents needs to be maintained in response to the change. All of the above seem to pose no insurmountable technical problems: the specification of relationships can be entered using standard interface widgets, diagrams can be constructed using simple graphical primitives and responding to changes in documents is can be reduced to a question of traversing a dependency graph, if one exists and submitting messages to any users affected. I therefore leave the details of this design to a later stage.

6.5 Architecture

In [10], Booch, Rumbaugh and Jacobson (the so-called "three amigos") define software architecture so:

> "An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization—these elements and their interfaces, their collaborations, and their composition."

This abstract formulation captures what is probably a common intuition about software architecture, namely that it has to do with the relationships between the components that make up a software system, and it does so in a way that stresses the engineering aspect of software architecture. But architecture can be interpreted differently, namely as the "style and method of design and construction [of a complex system]" [5], which is a definition closer to how conventional architecture can be perceived, namely as the consistent and reasoned application of design principles to achieve the most desirable balance between what is aesthetically pleasing, functionally required and economically feasible. I see the job of the software architect as much the same as that of the traditional architect, i.e. a person who design structures to be implemented by engineers and used by whomever contracted them or otherwise has use for them. The InSiter architecture is therefore designed to accomodate the needs of both parties by being:

- As simple as possible, but no simpler.
- Open and extensible.
- Flexible to allow for user customization and easy redesign at later stages.
- Robust, reliable, efficient and secure, in that order of significance.

6.5.1 Information architecture

I use the term "information architecture" to signify the choice and design of data structures to represent the information and interface components that the user can access through the InSiter interface. It is assumed that all data structures in the finished program correspond to interface components, although some may do so indirectly, perhaps by serving some auxiliary purpose in relation to another structure. The below only addresses the information architecture issues in InSiter that are either new, i.e. not addressed by PeerView, or of particular interest because they affect the entire design or pose some form of difficulty.

The information space multigraph

The most complex portion of the InSiter interface is the panorama where artifacts are placed and manipulated. The collection of artifacts visible in the panorama constitute a *multigraph*, i.e. a graph structure where pairs of vertices may be connected by multiple edges rather than a unique edge [2]. The vertices correspond to either *author nodes* or *document nodes* in the InSiter panorama and the edges correspond to relationships between authors, between documents or between author and document(s) [3]. Relationships may be either unilateral or bilateral and the direction of an edge between a pair of vertices indicates which apply in a given case. An example is given in Figure 6.7 where a set of author nodes are indicated by vertices marked An and a set of document nodes are indicated by vertices marked dn (with n being an integer).

[2]A multigraph can be defined as a pair $MG = (V, E)$ where $V \neq \emptyset$ is a finite set of vertices and E is a set of two-element subsets of V, where any $v \in V$ may be in more than one $e \in E$

[3]The terms vertex and node will be used interchangeable throughout this Chapter.

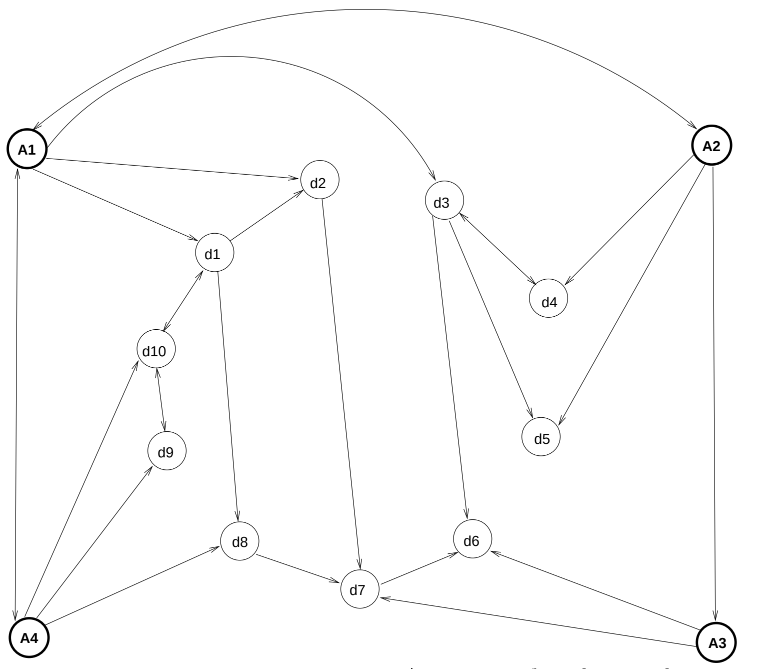


Figure 6.7: An example of an information space multigraph

One interpretation of the graph in Figure 6.7 could be that author $A1$[4] is the current maintainer of documents $\{d1, d2, d3\}$ and that authors $\{A2, A4\}$ are her co-workers. Similarly, author $A3$ is responsible for documents $\{d6, d7\}$, but she does not have a bilateral relationship to any of the other authors, and the unilateral relationship to author $A2$ is most likely an indication that $A2$ is her superior in the organization to which they all belong. The relationship between author and document is always unilateral and therefore represented by an edge directed from the author node to the document node. Documents, on the other hand, may be bilaterally or unilaterally related, as appears from Figure 6.7. A unilateral relationship may indicate a one-way dependency such as that between a `C++` code file and a standard library, or it may indicate some other, user-specified relationship such as that between a legal text and its annotations or a template and the documents based on it. A bilateral relationship may indicate a two-way dependency such as what exists between a `C++` definition file (typically suffixed by `.cpp`) and the corresponding declaration file (suffixed by `.h`), or it may indicate an entirely different bilateral relation between documents.

The multigraph structure described above is an abstract representation of what can be found in the InSiter panorama. The user will not need to be aware of this structure when placing artifacts in the panorama, although she might be, but it will be the underlying structure behind the artifacts visible in the InSiter interface. The concrete meaning of the multigraph will depend on how it is *annotated*, i.e. what properties the individual vertices and edges are "decorated" with as users collaborate and add artifacts to the panorama. For instance, the graph in Figure 6.7 is an abstract representation of the panorama that results when 4 authors collaborate in a group on a collection of 10 documents in all, but it says nothing about either the rendering of nodes or the rendering of relations between them. Let us assume that author $A1$ has added one text document containing HTML code and two bitmap graphics documents to the panorama, represented by nodes $d1$ and $d2, d3$, respectively. Let us further assume that

[4]Rather, the author designated by the node labelled $A1$, but I will dispense with such verbiage and simply refer to nodes as authors and edges as relationships where appropriate.

she has indicated that node $d1$ is related to nodes $d2, d3$ by a dependency relation since the latter two are bitmaps that are referenced by the HTML code in the former. Let us also assume that when $A1$ joined the group to which she currently belongs, she not only submitted her own profile to the group, but also stated her relationship with the other members, i.e. authors $A2, A3, A4$ in a similar manner to how she indicated the relationship between documents. We then have two sets of information that can be used to determine the rendering of both documents and author nodes as well the relationships between them. This gives rise to an *annotated multigraph* which can be defined as a set of five sets $AMG = \{V, E, A, Av, Ae\}$, where $V \neq \emptyset$ is a set of vertices, E a set of two-element subsets of V, A is a set of annotations, $Av = \{(v, a) \in Ae : v \in V \wedge a \in A\}$ and $Ae = \{(e, a) : e \in E \wedge a \in A\}$. An annotated multigraph is a both structural and databased representation of the information space that an InSiter panorama constitutes and therefore seems an appropriate data structure on which to base its design. This argument is made stronger by the added benefits it brings with it, namely a large body of research on graph related issues and algorithms and much existing code for implementing such structures.

Of course, as with any graph structure, the abstract, set theoretic definition above says nothing about how it should be embedded in the plane, or, in the case of a zoomable display, in three-dimensional space. This means that there is no inherent correlation between the abstract multigraph representation and the placement of the corresponding artifacts in the panorama. An illustration of this is in the relationship between Figures 6.1, 6.8 and 6.7. Figure 6.1 shows how the InSiter panorama might appear (these are tentative design plans, so final versions may differ) after four users have joined a group and added artifacts to the panorama. The number of nodes and relationships between them (indicated by lines) correspond to the number of vertices and edges in Figure 6.7, but there is no deliberate correlation between the (arbitrary) layout in Figure 6.7 and the deliberate layout in Figure 6.1. Figure 6.8 shows the relation between the two other Figures more clearly. It is a diagrammatic representation of the panorama where attributes have been removed to indicate only the graph structure underlying the panorama contents. Neither Figure 6.1 nor Figure 6.8 contain edges between documents as this would obscure these particular illustrations, but the intention is that users should be able to regulate freely which types of nodes and relationships should be rendered in the panorama. The absence of edges indicating relationships between documents could therefore be interpreted as the user having requested that such edges not be drawn. Section 6.4 has more on this topic and also shows how document relationships are shown elsewhere in the panorama, namely in the "Document context" area of individual document nodes.

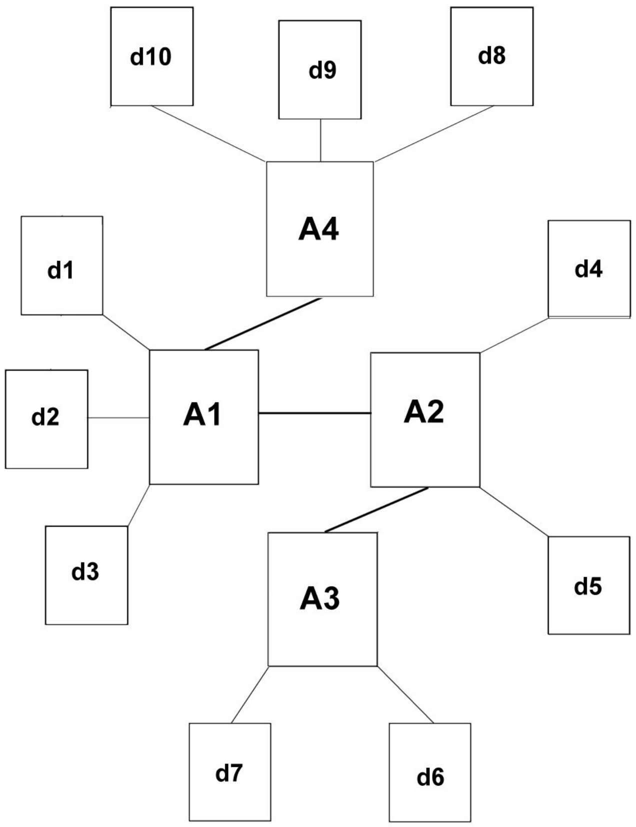


Figure 6.8: A diagrammatic representation of the panorama in Figure 6.1 to show its relationship to the multigraph in Figure 6.7. Note that the physical layout, which may have been applied manually by the user or automatically by a layout manager, has no deliberate relation to the layout of the graph in Figure 6.7 since a graph, by definition, has no inherent geometric embedding.

The multigraph structure will be represented by a modified version of the scenegraph structure that is used in the Jazz library as discussed in Section 3.3. Nodes in a Jazz scenegraph can be either leaves or inner nodes, both of which can be decorated with arbitrarily complex visual components which is the mechanism used for storing documents in PeerView. This corresponds to the annotations of the multigraph discussed above, and the relationships between nodes can be implemented by wrapping leaf nodes in a simple structure containing a collection of pointers to other leaves (or inner nodes) and tailoring algorithms that are either built from the ground up or taken from a third party library to work with this modified structure. Since the Jazz scenegraph supports many different types of operation, from translation, scaling and rotation to invisibility, hyperlinking and fisheye deformation, such a modified structure will also be endowed with a host of sophisticated graphics facilities at little or no extra cost in development time. The details of this design are not in place at the time of writing, but as the above suggests, it seems a both feasible and promising implementation model.

Data storage

InSiter will generate and need to store persistently (on disk) or transiently (in memory) the following types of data:

- Review contributions — these are plain text contributions by individual authors as well as the meta-data needed to identify and classify them (author name, date of creation and so forth).
- Review discussion structure — these are structural descriptions of the relationships between review contributions. They can converted into a textual representation since contributions are identified by their meta-data.
- Panorama structure — the multigraph defining the contents of the InSiter panorama can be converted to a pure text form ("serialized", in Java parlance) and hence easily stored in a text file.
- Document artifact statistics — as appears from Figure 6.5, InSiter will maintain statistical information on each document and since such statistics span the lifetime of the document, it will have to be committed to persistent storage between invocations of PeerView.
- Author information — this does not include the live video feeds of authors, but only encompasses their personal and other information as shown in Figure 6.4 which can all be converted to a pure text representation and thus easily saved to disk.
- Group data — the collaborative units that authors can form (referred to as groups at this early stage of design) are represented by textual data specifying name, description, participants and so forth, all of which can be stored as pure text.

As is suggested by the above description, I plan to store as much data as possible in a pure text form by using the Java serialization mechanism which allows data structures that implement a specific interface (the `Serializable` interface) to be converted to a textual representation and to subsequently be restored from one. By doing so, InSiter can commit all of the above data to disk using the standard stream facilities found in the JDK [50] when the application is terminated, and restore them from disk when it is subsequently restarted. This approach is used in PeerView which saves discussion fora, discussion contributions, group directories and preferences to disk as serialized representations which is a simple and compact representations that can be manually inspected using standard text processing tools. This might be useful if scouring the materials generated for a specific snippet of information such as a remark in a discussion contribution. InSiter will reap those benefits by using the text representation discussed above, and it can improve on PeerView's storage method by using (optional) compression of files above a certain size which will of course make the text files inaccessible to simple manual inspection, but will minimize the InSiter disk "footprint"[5] if so desired.

Further elaboration of the design might make it necessary to introduce new data structures, but I intend to make them serializable as well so as to not introduce exceptions to the above scheme. I expect little difficulty in doing so since any data structure can be described as a

[5]The amount of storage allocated.

collection of elements and relations between them[6] which ought to be amenable to textual representation.

6.5.2 Distributed architecture

As described in Section 5.3.4, the Light-weight Reliable Multicast Protocol (LRMP) package from the French research institute INRIA seems a promising candidate for implementation of a multicast distribution architecture which in turn seems a good choice of such an architecture for colloborative systems. Indeed, in [37], Liao reports that the LRMP has been used in such applications previously and further describes the design goals of the LRMP library which have been to deliver a reliable protocol for bulk transfer of data, precisely what is needed in InSiter's case. The LRMP is a transport protocol meaning that it implements the transport layer of a network protocol and therefore does not aim to support what is typically left to layers above the transport layer, namely authentication and session management services. This need not be a problem because the LRMP can be used in conjunction with the *WebCanal* [39] package, also from INRIA, which provides high-level primitives for creating and maintaining channels that can be used as communication conduits by the colloborative units. An alternative solution to using the LRMP protocol could be to build on top of the `MulticastSocket` class that is part of the standard JDK [50]. This class offers the bare minimum of services needed to send and receive multicast data, but can be extended using the standard Java extension mechanisms (composition and inheritance) so one could conceivably extend this class to support both managed groups and multiple channels in a manner similar to the JSDT. The advantage of that approach would be that one could customize freely the multicast protocol to suit the particular needs of InSiter which would most likely yield better performance than using a third-party solution aimed at generalized support for applications relying on multicast. Also, it would reduce the potential for complication incurred by interfacing technologies to an absolute minimum since extending a JDK class to fit into the InSiter framework would not constitute a technology interface, but simply a program component like any other.

It therefore seems entirely feasible to implement the InSiter architecture based on a multicast protocol as an implementation can be obtained over the Internet or built from the ground up using tested and well-known technology. Issues of security and reliability will have to be dealt with separately as neither the LRMP or the `Multicastsocket` class have built-in functionality to support those aspects. However, the Java JDK contains several classes for implementing security layers and encryption technology for Java is freely available over the Internet (see for example [13]), so fitting a distributed architecture with customized security and encryption layers seem no more complicated than implementing the multicast transport layer.

6.6 System requirements

InSiter is to be implemented in Java. This is a given if the Jazz library is to be used since interfacing with a different language brings with it its own set of difficulties. I do not expect performance difficulties to hamper usability or development since PeerView has demonstrated

[6]This may sound like a gross generalization, but consider for a moment that a "data structure" by definition is a collection of discrete symbols sets and a collection of relations (which can be defined as pairs of elements) between them, and the statement may seem more plausible.

that both the implementation of a zoomable interface and the distribution of data, even using a non-optimal delta algorithm (see Section 3.3.6), do not offer unsurmountable performance challenges. Nor do I expect the added layer of data incurred by the introduction of video transmission to degrade performance significantly as suggested by the calculation on page 61, since the frame rate can be adjusted according to available bandwidth and compression can be applied to further reduce transmission volumes. The feasibility of the former measure is documented in for example [17], and typical compression ratios of video data can be very good indeed as is demonstrated by the popular video formats AVI and MPEG. I therefore expect the minimum requirements for InSiter to be about the same as those for PeerView, i.e. an average PC system as found in many private homes and in many offices. Each machine should have a network connection in order to participate in groups, and if users wish to use the video facilities, their machines should be equipped with a camera which need not cost more than a few hundred dollars today, depending on resolution and colour range. These modest system requirements, coupled with the choice of Java as implementation language, should make InSiter available to a wide audience and help minimize the deployment difficulties that sometimes arise when new systems are shipped to users.

Chapter 7

Conclusion

7.1 Results and contributions

The work conducted in connection with this thesis has produced the PeerView software which has been the subject of two expositional articles, one quite short which appeared in the February 2001 issue of IEEE DSOnline [61] and another, somewhat longer article coauthored with Jyrki Katajainen which is included in this thesis in Chapter 2.4. The latter article has also appeared as a DIKU technical report [62] and is currently under review by an international journal. PeerView is listed in various directories of applications demonstrating groupware concepts and zoomable interface technology, and the visitor statistics I have gathered for the PeerView home page indicate that it has been frequently accessed by visitors from around the globe, many of whom seem to have downloaded the application.

In the objective terms of public exposure and amounts of code written, PeerView can be said to have fulfilled its purpose of being a prototype vehicle for artifact rendering and group review. Also, the experimental evaluation results and analysis contained in this thesis can likely help successor system(s) be developed more efficiently and to a higher standard than if they had had to be developed without any such empirical base. I believe much of this experience can be of interest also to developers of otherwise unrelated forms of groupware since Chapters 3 and 4 contain information on such matters as distribution technology, design criteria and evaluation techniques that may help developers make appropriate design and implementation choices. An outcome from PeerView that is of potential interest both to groupware developers and to developers building single-user applications is the experience using the Jazz and JSDT component technologies to shorten development time and the deliberations made in choosing them, as discussed in Chapter 3. The PeerView source code is in a sense the definitive reference to anyone who wants to study the details of how these third-party technologies have been used and it has therefore been made public for anyone to peruse.

PeerView has contributed to existing research literature on common information spaces by addressing recurring problems with largely untried techniques, such as using a zoomable display in response to the "detail/overview" problem as discussed in Section 2.2, and a threaded discussion forum for generating and organizing dialogue among users.

7.2 Lessons learned

On a personal level, PeerView has helped me develop my skills as a developer by being a non-trivial development project in territory that was largely untried for me when I started. It has also helped improve my research skills since it spans such a broad range of topics, ranging from theoretical CSCW research on common information spaces [4] to delta algorithms [31] and APIs [50, 34]. The development, research and writing processes have taught me new things about project logistics and confirmed what I already knew or suspected, namely that I tend to be overly optimistic when planning such projects and that as schedules slip, difficult and unpleasant choices have to be made by removing or scaling down planned features and lowering ambitions.

The methodology of building a prototype from an initial idea, experimentally evaluating the end product and performing an analysis of the development efforts to help guide future development proved fruitful in that it did yield useful information and did produce functional software that tested ideas and technologies in practice. Further, the PeerView software provides a "prototyping base" for similar systems since it allows developers to incorporate ideas into an existing framework, i.e. the PeerView source code, and subject them to experimental testing before making end-user versions of them.

This does not mean that the methodology described in Section 2.3 was followed with unswerving loyalty nor that it cannot be improved. For example, as appears from for example the discussion in Section 3.3.6 and 3.3.1, not all decisions were planned for or considered as thoroughly as could be desired. Further, not all phases and steps of the development process took place in the prescribed order. I believe the initial design could have been more stable if I had spent more time studying reports from similar initiatives such as [20] and [7] and drawn the appropriate lessons from the experience reported therein. An example of how this affected the process in practice is reported in Section 3.1 where I only became aware of research on the "detail/overview" display problem after I had carried through my own reasoning on the issue. The experimental evaluation described in Chapter 4 was not planned in detail until after implementation was concluded and the software made public, i.e. quite late in the work process, which resulted in it being limited to the number of participants I could recruit with short notice. In retrospect, I believe that experimental evaluation would ideally be an ongoing series of experiments that punctuated different stages of development and were planned as such, i.e. in detail from the beginning of the project. The first experiment could then be an assessment of the interface design in isolation based on the "mock-up" made during the early stages of design. The second experiment could be conducted when core functionality[1] was implemented and subsequent experiments as additional functionality was added and official releases made. This would of course incur greater costs in terms of time taken to plan experiments, recruit participants and possibly compensating them economically, but the feedback into the development process would make this well worthwhile, I suspect. The techniques used for experimental evaluation worked well in this case and are widely used as documented in [45], but if developers lack the resources to regularly schedule such experiments, they could probably benefit from using a form of heuristic evaluation tailored for groupware [3] for evaluation at the early stages of development.

[1]In PeerView's case, this might adding and removing documents and manipulating the group directory.

7.3 Future work

If possible, I intend to direct future work at completing the design of InSiter which was begun in Chapter 6 through an iterative development process where successive design increments are implemented and the resulting intermediate systems subjected to evaluation which feeds back into the ongoing development process. This would, I believe, be a fruitful methodology for realising a system that could go beyond the PeerView basics to provide users with a useful, scalable and generically designed support system for collaborative work centered around a common information space. However, for this to move beyond wishful thinking there must be sufficient funding in place as InSiter is considerably beyond what can be comfortably be implemented by one or two developers in their spare time. Depending on investor sentiment, InSiter could then become a freely available system for a general user audience or a commercial, licensed system for specific customer groups. Either way, the development process in itself would probably be worth the effort since it can, as demonstrated in this thesis, be a useful asset *per se*.

Bibliography

[1] P. Young (Editor), *3D Software Visualization — Visualisations and Representations*, Worldwide Web Document (1997). Available at `http://vrg.dur.ac.uk/misc/PeterYoung/pages/work/documents/BT-report/`.

[2] J. Arvo (Editor), *Graphics Gems II*, Academic Press, Inc. (1991).

[3] K. Baker, S. Greenberg, and C. Gutwin, *Heuristic Evaluation of Groupware Based on the Mechanics of Collaboration*, Technical report, Department of Computer Science, University of Calgary (2000).

[4] L. Bannon and S. B. dker, Constructing Common Information Spaces, *Proceedings of EC-SCW97, Dordrecht*, Kluwer (1997), 81–96.

[5] P. L. P. M. Barbara Hayes-Roth,Karl Pfeger and M. Balabanovic, A Domain-Specific Software Architecture for Adaptive Intelligent Systems, *IEEE Transactions on Software Engineering* **21**,4 (1995).

[6] K. Beck, Extreme Programming Explained (2000).

[7] B. Bederson and J. Meyer, Implementing a zooming User Interface: experience building Pad++, *Software Practice and Experience* **28**,10 (1998), 1101–1135.

[8] B. B. Bederson and B. McAlister, *Jazz: An Extensible 2D+Zooming Graphics Toolkit in Java*, Technical Report CS-TR-4015, University of Maryland, College Park (1999).

[9] A. F. Blackwell, *Metaphor in Diagrams*, Ph. D. Thesis, Darwin College, University of Cambridge (1998).

[10] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 1 Edition, Addison-Wesley, Reading, Massachusetts, USA (1999).

[11] F. P. Brooks, No Silver Bullet - Essence and Accidents of Software Engineering, *IEEE Computer* (1987), 10–19.

[12] Centrinity Inc., *Centrinity*, Website accessible at `http://www.softarc.com`.

[13] www.cryptix.org, Website accessible at `http://www.cryptix.org`.

[14] OpenAvenue, Inc., *Concurrent Versions System: The Open Standard for Version Control*, Website accessible at `http://www.cvshome.org`.

[15] Cybozu Inc., *Cybozu*, Website accessible at `http://www.cybozu.com`.

[16] L. P. Deutsch, *RFC 1951: DEFLATE Compressed Data Format Specification version 1.3* (1996). Status: INFORMATIONAL.

[17] P. Dourish and S. Bly, Portholes: Supporting Awareness in a Distributed Work Group, *ACM Annual Conference on Human Factors in Computing Systems* (1992).

[18] Lehrgebiet Praktische Informatik II, FernUniversität Hagen, *DreamTeam Homepage*, Website accessible at `http://carmen.fernuni-hagen.de/dreamteam/dreamteam_eng.html`.

[19] S. Easterbrook, *Coordination Breakdowns: Why Groupware is so Difficult to Design*, Technical Report CSRP 343, School of Cognitive and Computing Sciences, University of Sussex.

[20] G. W. Furnas and X. Zhang, MuSE: A Multiscale Editor, *Proceedings of the ACM Symposium on User Interface Software and Technology*, *Zoomable User Interfaces* (1998), 107–116.

[21] R. L. Glass, *Software Runaways: Monumental Software Disasters*, Prentice Hall, Upper Saddle River (1998).

[22] S. Greenberg, C. Gutwin, and A. Cockburn, Awareness Through Fisheye Views in Relaxed-WYSIWIS Groupware, *Graphics Interface '96*, W. A. Davis and R. Bartels (Editors), Canadian Human-Computer Communications Society (1996), 28–38. ISBN 0-9695338-5-3.

[23] R. E. Grintner, Workflow Systems: Occasions for Success and Failure, *Computer Supported Cooperative Work* **9** (2000), 189–214.

[24] Department of Computer Science, University of Calgary, *GroupLab: Laboratory for Computer Supported Cooperative Work & Human Computer Interaction*, Website accessible at `http://www.cpsc.ucalgary.ca/projects/grouplab`.

[25] Diamond Bullet Design, *Types of Groupware and Groupware Issues*, Worldwide Web Document (2001). Available at `http://www.usabilityfirst.com/cscw.html#TypesofGroupware`.

[26] C. Gutwin and S. Greenberg, Effects of Awareness Support on Groupware Usability, *Proceedings of ACM CHI 98 Conference on Human Factors in Computing Systems*, *Usability of Groupware* **1** (1998), 511–518.

[27] C. Gutwin and S. Greenberg, *A Framework of Awareness for Small Groups in Shared-Workspace Groupware*, Technical Report 99-1, Department of Computer Science, University of Saskatchewan (1999).

[28] National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, *NCSA Habanero*®, Website accessible at `http://havefun.ncsa.uiuc.edu/habanero/`.

[29] D. T. Hoang, *Fast and Efficient Algorithms for Text and Video Compression*, Technical Report CS-97-06, Brown University - Department of Computer Science (1997).

[30] W. S. Humphrey, A Personal Commitment to Software Quality (1995), 5–7.

[31] J. J. Hunt, K. P. Vo, and W. F. Tichy, An Empirical Study of Delta Algorithms, *Software configuration management: ICSE 96 SCM-6 Workshop*, I. Sommerville (Editor), Springer (1996), 49–66.

[32] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley (1999).

[33] I. Jacobson and others, *Object-Oriented Software Engineering — a Use Case Driven Approach*, Addison-Wesley (1992).

[34] Human-Computer Interaction Lab, University of Maryland, *Welcome to Jazz!*, Website accessible at `http://www.cs.umd.edu/hcil/jazz/`.

[35] C. A. Jones, The Java Internationalization API: Global software for the global village, *Dr. Dobb's Journal of Software Tools* **23**,1 (1998), 54, 56–69, 103–104.

[36] Sun Microsystems, Inc., *Java*$^{\text{TM}}$ *Shared Data Toolkit*, Website accessible at `http://java.sun.com/products/java-media/jsdt/index.html`.

[37] T. Liao, *Light-weight Reliable Multicast Protocol*, Technical report, INRIA (1998).

[38] Lotus Development Corporation, *Lotus Notes*, Website accessible at `http://www.lotus.com/home.nsf/welcome/notes`.

[39] Inria, *LRMP Home Page*, Website accessible at `http://webcanal.inria.fr/index.html`.

[40] S. McConnell, *Code Complete*, Microsoft Press (1990).

[41] B. Meyer, Component and Object Technology: On to Components, *Computer* **32**,1 (1999), 139–140.

[42] H. D. Mills, M. Dyer, and R. Linger, Cleanroom Software Engineering, *IEEE Software* **4**,5 (1987), 19–25.

[43] J. Nielsen, *Usability Engineering*, Academic Press, Inc. (1993).

[44] www.opengl.org, *Open GL*, Website accessible at `http://www.opengl.org/`.

[45] D. Pinelle, *A Survey of Groupware Evaluations in CSCW Proceedings*, Technical report, Department of Computer Science, University of Saskatchewan (2000).

[46] M. Roseman and S. Greenberg, TeamRooms: Network Places for Collaboration (1996), 325–333.

[47] C. G. Saul Greenberg and A. Cockburn, Using Distortion-Oriented Displays to Support Workspace Awareness, R. C. A.Sasse and R.Winder (Editors), Springer-Verlag (1996), 299–314.

[48] K. Schmidt and L. Bannon, Taking CSCW seriously: Supporting articulation work, *Computer Supported Cooperative Work* **1** (1992), 7–40.

[49] B. Schneiderman, *Designing the User Interface*, Addison-Wesley (1998).

[50] Sun Microsystems, Inc., *Java*$^{\text{TM}}$ *2 SDK, Standard Edition*, Website accessible at `http://java.sun.com/products/jdk/1.2/`.

[51] I. Sommerville, *Software Engineering*, 5th Edition, Addison-Wesley Publishing Company, Inc. (1996).

[52] J. Stasko, J. Domingue, M. H. Brown, and B. A. Price (Editors), *Software Visualization: Programming as a Multimedia Experience*, The MIT Press (1998).

[53] WebWisdom.com, Inc., *TANGO Interactive*$^{\text{TM}}$, Website accessible at `http://www.webwisdom.com/tangointeractive`.

[54] Lars Yde, *The PeerView Website*, Website accessible at `http://www.diku.dk/research-groups/performance-engineering/PeerView/`.

[55] E. Tufte, *Envisioning Information*, Graphics Press (1990).

[56] Diamond Bullet Design, *Usability First*$^{\text{TM}}$, Website accessible at `http://www.usabilityfirst.com/`.

[57] WebMagic, Inc., *USENET.org*$^{\text{TM}}$, Website accessible at `http://www.usenet.org`.

[58] IBM, *VisualAge*, Worldwide Web Document. Available at `http://www-4.ibm.com/software/ad/vajava/`.

[59] A. Woodruff, J. Landay, and M. Stonebraker, Goal-Directed Zoom, *Proceedings of ACM CHI 98 Conference on Human Factors in Computing Systems (Summary), Late Breaking Results: Look and Learn: Visualization and Education Too* **2** (1998), 305–306.

[60] Distributed Systems Technology Centre Pty. Ltd., *The wOrlds Project*, Website accessible at `http://archive.dstc.edu.au/TU/wOrlds/`.

[61] L. Yde, PeerView - a Prototype for Rendering and Review, *Distributed Systems Online* (2001).

[62] L. Yde and J. Katajainen, *Supporting Intellectual Work Through Artifact Rendering and Group Review*, Technical Report 00/11, Department of Computer Science, University of Copenhagen (DIKU) (2001).