The eFacilitator: A Meeting Capture Application and Infrastructure

by

Harold Fox

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2003

© Massachusetts Institute of Technology 2003. All rights reserved.

Author

Department of Electrical Engineering and Computer Science

September 30, 2003

Certified by

Howard E. Shrobe
Principal Research Scientist
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Students
The eFacilitator: A Meeting Capture Application and Infrastructure

by

Harold Fox

Submitted to the Department of Electrical Engineering and Computer Science on September 30, 2003, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering

Abstract

Meeting capture and support is an important research field in ubiquitous computing and human computer interaction. We have built a note-taking application, which lets meeting facilitators and note takers organize the material they place on an electronic palette. This application stores all transactions, so that the note-taking process can be played back in real time along with the captured audio and video. Eventually, this captured note organization can be used for topic searches and information extraction. We have also built a browsing program to let users find meetings based on properties like time and location. This program can also create meetings which go into the same data space. The data space lets agents representing a meeting automatically find capture hardware to capture and index meetings without user intervention. The data space is enabled by a highly scalable distributed, object-oriented, semantic database.

Thesis Supervisor: Howard E. Shrobe
Title: Principal Research Scientist
Acknowledgments

I would like to thank my advisor, Howie Shrobe, for always listening to what I had to say, no matter how off topic it was. Further thanks go to Kimberle Koile and to my office-mates, Max Van Kleek, Gary Look, and Aaron Adler, for their intelligent conversation and encouragement. I would also like to thank Acer for their donation of fabulous tablet computers and to Microsoft for putting their resources behind tablet computing.
Contents

1 Introduction .............................................. 11

1.1 Motivation for Meeting Capture: Knowledge Management ....... 12
1.2 Approach .............................................. 13
  1.2.1 Effective Meeting Facilitation .......................... 14
1.3 eFacilitator Overview .................................... 15
1.4 Thesis Summary ......................................... 16

2 Use Scenarios ............................................ 19

2.1 The Class Lecture ......................................... 19
2.2 A Brainstorming Meeting .................................. 21
2.3 Product Triage Meeting .................................... 22
2.4 Military Mission Planning .................................. 23

3 System Description ...................................... 25

3.1 Recommended Hardware Infrastructure .......................... 25
3.2 Object Browser .......................................... 26
3.3 The Agenda Manager ....................................... 30
3.4 The eNotePad ........................................... 31
  3.4.1 Previous Versions ..................................... 31
  3.4.2 Interface Design Principles for Tablet PC's .................. 33
  3.4.3 High Level Goals: Organize, Associate, Annotate .............. 34
  3.4.4 eNotePad Detailed Features ............................. 36
3.5 The NotePlayer .......................................... 41
4 Implementation

4.1 Scheduling and Finding Meetings ................................................. 43
  4.1.1 The Object Server ....................................................... 44
  4.1.2 The Object-Oriented Database ........................................ 44
  4.1.3 The Semantic Database .................................................. 52
  4.1.4 The Distributed Semantic Database ..................................... 53
  4.1.5 The Object Browser ....................................................... 60
  4.1.6 Channels ................................................................. 63

4.2 eNotePad Implementation: The Geometry of Handwritten Notes ........ 64
  4.2.1 Geometries ................................................................. 65
  4.2.2 Sketches ................................................................. 68
  4.2.3 Notes ................................................................. 70
  4.2.4 The Shared eNotePad ..................................................... 70
  4.2.5 The NotePlayer .......................................................... 71

5 Previous Work ............................................................................. 73

5.1 Evolution of Electronic Conference Rooms .................................... 73
5.2 Evolution of Capture and Retrieval Tools ...................................... 75
5.3 Evolution of Collaborative Note-Taking Software .............................. 76
5.4 Evolution of Meeting Automation .............................................. 78

6 Conclusions and Future Work ...................................................... 81
List of Figures

3-1  Layout for the Object Browser ........................................ 26
3-2  First eNotePad prototype: Users create an outline of ideas ........ 32
3-3  The latest eNotePad: Users can move notes around and arrange them in a hierarchy ........................................ 32
3-4  Three types of visual representations people use to convey meaning 35
3-5  The eNotePad's context menu ........................................ 38
3-6  An example of the splitting operation ................................ 38
4-1  Finding the points of an index that match a query set .............. 50
4-2  Six databases in a hierarchy of scope ................................ 55
4-3  Basic Model View Controller Architecture .......................... 65
4-4  Advanced Model View Controller Architecture ........................ 66
Chapter 1

Introduction

Collaboration and the field of ubiquitous computing have a symbiotic relationship. As technology has advanced, business processes have become ever more complex, interconnected, and dynamic. This complexity demands an increase in collaboration among workers with different backgrounds and abilities. At the same time, the new flexibility of the labor force has made collaboration more difficult as employees are increasingly asked to work with people they don’t know, who may not even be collocated in the same place.

While collaboration has gotten more difficult, computing and information technology have gotten easier. The new mantra for human computer interaction researchers is “ubiquitous computing.” This movement seeks to place networked computers and sensors everywhere. The ultimate goal is for computers to understand everything going on in the world and to provide the user with exactly what they need wherever they are. While it presents a powerful vision, ubicomp is struggling to find a “killer app,” an application that compels customers to upgrade their technology, spurring a virtuous cycle of technology investment and social advance.

With the above mentioned changing nature of work, businesses will realize large efficiency gains by improving collaboration. We believe that this is the area where ubicomp can make its biggest economic impact.

In this thesis, we describe the eFacilitator, a system that provides meeting capture and support in intelligent conference rooms. Real-time meetings are the most pow-
erful engines of collaboration, whether structured or unstructured, local or remote. The eFacilitator provides support for meetings, using the best practices well known to professional facilitators and organizers. More importantly, it provides meeting capture, potentially a powerful mechanism for knowledge management.

1.1 Motivation for Meeting Capture: Knowledge Management

To address the emerging needs of organizations, management researchers have developed and encouraged the practice of knowledge management. The idea is that the assets of a modern company are the specialized knowledge, heuristics, and best practices inside the heads of its employees. The more widely this knowledge is spread through the organization, the more productive it is. Organizations need to encourage a culture of knowledge sharing. Senior employees should not fear for their livelihood if they reveal trade secrets to more junior colleagues. They should be rewarded for taking the time to write educational documents. Finally, employees should not be afraid to reveal that they do not understand something, so they can actively seek out people in the organization who may know how to solve their problem.

Capturing meetings is a cost-free way to improve knowledge management with unlimited potential. When a group of experts talk about how to solve a problem, their interaction is automatically captured, stored, and indexed. When someone else encounters a similar problem, they can review the previous discussion to get ideas and context.

If a designer is asked to build on some previous design or implementation, there will often be many tricky or incomprehensible pieces. S/he can go to the meeting records for the previous design team. Very likely, the tricky and non-standard aspects of the design will be spelled out and talked about in the design meetings and design reviews.

Finally, with meetings captured, certain large, political decisions can be made
transparent. The affected people can see the opinions of all of the participating parties, and they can see which opinions prevailed. Understanding the direction of top leadership and the rationale for big initiatives will make employees feel empowered and motivated.

1.2 Approach

Meeting support and capture can be studied from many different angles. Some research groups have focused on the pure recognition problems: Who is speaking? What are they saying? Who are they talking to? What are the segments of the dialogue? Other groups have focused on the human interface problems. How can computers replace notepads and whiteboards to enable capture and sharing without decreasing usability?

We have chosen the latter path, treating meeting capture as a human interface problem. In other words, how can we build an application that is so friendly and responsive that it convinces human users to tell it what is taking place in a meeting?

This is more effective than trying to do raw recognition from the highly unconstrained audio-visual input. Schultz et al. have attempted meeting transcription with participants wearing close, lapel microphones with an acoustic model trained on ten hours of similar speech [24]. Even with these constraints, the speech recognition system still has word error rates of at least 38.7%. This percentage is far too high to extract useful information from a transcript, and it is also unreasonable to expect users to bother with clip on microphones and wires when they want to have a face-to-face meeting. For audio capture in our prototype conference rooms, we use area microphones on tables and in the ceiling. Humans have no trouble understanding speech from these sources, while state of the art speech systems perform quite poorly.

Instead of using audio and video to understand meetings, we choose to focus on the notes that users take, and we implement a pen-based application to let users take notes and organize them. We depart from the most recent work in note-taking applications by explicitly encouraging meeting practices like the Interaction Method
of Doyle and Straus [7]. This method, like other professional facilitation guidelines, assigns a structure to medium and large size meetings.

1.2.1 Effective Meeting Facilitation

Meeting facilitation is a skill that employees must now have at many companies. The essential task is to keep a large group focused on an agenda, enforce the meeting rules, prevent a small number from dominating the discussion, and encourage participation from less outspoken people. Facilitators do this by writing down the main points of everything that is said and trying to relate everyone's point to existing points on the board. If someone mentions something off topic, the facilitator puts it in the parking lot, a set of tangential issues not presently relevant, but could become agenda items for subsequent meetings.

While this structure may seem restrictive, it can actually foster much creativity if the agenda calls for it. For example, if the meeting's goal is to come up with a theme for a holiday party, the facilitator would ask for suggestions from the audience, and every idea that is thought up would go on the board. Seeing everyone's ideas together inspires the other meeting participants, creating a chemistry of creativity.

Good facilitation also aids conflict resolution. The facilitator summarizes all of the opposing opinions with all of the supporting arguments for each. Doing this, the facilitator ensures that s/he comprehends the key points of each position, ensuring that the group understands them as well. Then, the group can truly make an informed decision, since they know the tradeoffs of every option.

Generally, a meeting's public display serves as the group's working memory. Since humans can only hold about seven distinct items in their head at a time, this external memory is very important to keep a meeting on topic and keep all of the participants lively. In a sense, the goal of the eFacilitator is to enable the manipulation of this working memory and to capture it for future perusal. It captures exactly the structure of the meeting that we are trying to get for retrieval: What problems were considered? What decisions were made? What ideas were considered?

The earliest work in electronic meetings actually focused exactly on this point,
and it seems to have been lost in the technology driven applications of recent times. With the CogNoter program from the CoLab [27], meeting participants type their ideas into a keyboard. They would link these ideas together, and they could type an argument for or against an idea by clicking on it.

Theoretically, this is exactly the way meetings should work. However, humans' thoughts and interactions just do not emerge this precisely. They can be carefully organized and categorized only once they have come out in some way. Therefore, to support human thinking processes, we need to support more natural and fluid expression than a keyboard and mouse. Specifically, we have put a lot of energy into building a meeting support tool for primarily pen input. This is an important consideration. It makes our capture task enormously more difficult for possibly no good reason. However, we believe if a keyboard/mouse note taking program were as effective as a whiteboard or a spiral notebook, users would have adopted them in large numbers by now. With a writing surface, humans can draw anything anywhere. They can intersperse sketches with words, and they can draw many different kinds of boxes, arrows, and lines to reflect the mental structure they may have. We decided that we must support this freedom to garner user acceptance.

To summarize, we are exploring meeting capture by providing a tool for creating and organizing the points of a meeting with an intuitive pen interface.

### 1.3 eFacilitator Overview

The eFacilitator is comprised of four programs.

- The eNotePad is an electronic notebook with support for organizing, associating, and annotating notes.

- The NotePlayer is a meeting browsing program that synchronizes notes from the eNotePad with captured audio and video.

- The Agenda Manager is the program responsible for managing the meeting
room, setting up the capture streams, and sharing meeting materials with participating computers.

- The Universal Inbox is a structured search program to locate meetings with a variety of properties. It also contains an inbox for receiving notifications from personal agents.

1.4 Thesis Summary

In this thesis, we describe the interface and implementation of these programs. The project contributes three advances to previous work in meeting and collaboration support. The first is a distributed agent infrastructure that can set up and run meetings in a variety of spaces and configurations. The infrastructure lets meeting management and capture happen automatically without any technical intervention from users. They schedule a meeting for a certain room, and they are done. The second contribution is a new style of note-taking program, the eNotePad that lets meeting participants share notes and documents, and manage them on the public screens. The notepad specifically encourages the working memory manipulation that we have described above. The final contribution is the meeting browsing interface, a scheme to help users find the meetings and the parts of meetings that interest them. Users browse for meetings by searching for high-level attributes such as the time, place, and people involved. They can find a particular position in the meeting by quickly browsing through the notes and finding the note that indexes the discussion they were looking for. From here, they can play back the meeting in real time, listening to the discussion, and possibly even seeing a video.

Chapters 2 and 3 describe the user interface of the system's components and the rationale for each of the features. Part 4 describes some interesting points of the system implementation. Large sections focus on a distributed object-oriented database that enables flexible meeting automation and the geometry operations that lie behind the manipulations of notes. Part 5 presents a history of previous work in meeting support and capture. Part 6 concludes with a discussion of future work.
The full eFacilitator is currently under development. Not all of the features described here have been implemented. A prototype of the eNotePad has been written, which allows the creation, annotation, and sharing of notes. The Universal Inbox and Agenda Manager have not been implemented, although the object database to enable the automation has been built. The NotePlayer, for browsing a meeting record, is fully implemented.
Chapter 2

Use Scenarios

Our system is complex, with features that allow it to be deployed in many settings. To motivate the concepts we introduce and the features we implement, we will describe four scenarios that use the eFacilitator in different ways. The Class Lecture shows the capabilities of the Object Browser, the Agenda Manager, and the networking properties of the eNotePad. The other three scenarios show the collaborative benefit of the eNotePad in different social contexts.

2.1 The Class Lecture

It's 1:30: time for Professor Jones to prepare for his 2:00 class. He goes to his Object Browser and searches for events for today in the room NE43-518. He sees his class there at 2:00, where his scheduling agent automatically put it. He clicks on the event. In the event's view pane, he clicks on the agenda. There is a button there to add a presentation document to the main agenda item. He clicks on this button, and this brings up a new Object Browser window for browsing files. He finds the file in a subdirectory of his local directory and selects it. The file browser automatically copies the slides to a public, web accessible directory. He now sees his PowerPoint slides listed on the agenda.

Meanwhile, Bill, a student, is about to go to class. On his main computer, he receives a notification from his agent, reminding him of the appointment. He picks
up his tablet computer and heads out. On the way out, he realizes he has forgotten which room the class is in. This is not a problem, because the tablet always has the same information as his desktop. He goes to his calendar, finds the class at 2:00 and sees the room, NE43-518.

When Professor Jones arrives in the room, he takes his place at the front. His tablet has a location-detection device, so it knows it is in room 518. Using his location context, his inbox agent has put his class in the top slot. He clicks Start, and the starting slide of his lecture appears on the main projector. Also, two blank writing areas are projected to the left and right for him to develop ideas spontaneously.

After the professor has put the slides up, the students go to their own inboxes, find the class, and click the "Join" button. The eNotePad starts on their computer with the first slides and each of the two writing areas separated by tabs.

He gives his lecture. The slides contain the high-level, important points he wants to get across. At certain points in the lecture, he needs to develop certain equations and solve problems from the lecture in more detail. He does these first on the left board and then on the right board when the left becomes full. To keep the class engaged, he periodically asks questions to test their understanding. The students write their answers on their tablets, and they all make them public at once. The professor sees the responses and uses the feedback to decide if the students have mostly understood, and he can move on.

Bill, the student, has trouble with some of the points the professor brings up. Some of the equations are unclear, because the professor skipped some steps. He clicks on them, and then clicks on the annotate function. He creates space for himself under the equations and works through them in detail. Satisfied, he continues listening. On another occasion, he is not able to resolve his difficulty. He annotates the equation with a question. Bill raises his hand. He asks the question verbally and makes his written comment public. The professor answers, and Bill is satisfied.

At the end of the lecture, Professor Jones goes to his inbox. The class is still there in the list. He clicks "Stop", and all of the projectors turn off. The students finish writing their notes and close their eNotePads. The professor goes back to his
office, knowing that the whole lecture has been automatically captured, stored, and indexed.

Later, Bill is doing his problem set. He wants to review the class material from a week ago which is covered in the problem he is working. He goes to his Object Browser and looks through his calendar. He narrows the event list by restricting the event type to classes. Now, a new selection box appears for the name of the class. He clicks on the box, and he gets a listing of all of the classes he has been in recently. He finds the one from a week ago, clicks on it, and then selects View. The NotePlayer starts, showing the first slide of that lecture. Bill moves the scroll bar to the right, which rapidly flips through the slides and the notes Bill made on them. He finds the slide he was interested in. He clicks the play button, and he hears the professor's voice playing in sync with the slide view. He remembers the technique the professor showed and returns to his problem set.

Later on in the problem set, Bill finds a problem that requires knowledge of a concept taught in a prerequisite. Not having taken that class, Bill is worried. He goes to his Object Browser and searches for the class in the previous semester. He finds the syllabus and sees the date where the concept is covered. He uses the Object Browser to find the class that took place on that date. He views the lecture and listens to that class's instructor explain the concept. Bill listens and learns what he needs to solve the problem.

### 2.2 A Brainstorming Meeting

This scenario shows the eNotePad in a brainstorming meeting. It shows how its organizational capabilities can facilitate creativity as each concept written down helps to stimulate further ideas from the group.

Edusoft, an educational software company is trying to decide what to do for the next version of its Homework Helper product. Its current product lets students organize their classes, schedule their homework assignments, record their grades, and organize their electronic work.
The product development team has scheduled a meeting of the designers and developers to brainstorm a list of possible features for the next version. The meeting is fast and loose with no ideas thrown away or criticized. The first suggestion someone makes is to move the homework calendar feature from the menu to the toolbar to make it more visible. The facilitator writes this down on the electronic whiteboard. Someone else suggests creating a service so that homework assignments can be automatically downloaded from the web.

Following this suggestion, another person mentions the feature of receiving grades from the web as well as turning in assignments on-line. Noticing the similarity of all of these features, the facilitator combines them into a group and puts them under the category "Networking/Semantic Web". She creates another category, "Desktop Improvements," to place the other features. From this grouping, a product manager suggests creating a whole new product: Homework Helper, Teacher Edition.

The meeting continues, producing a long list of possible ideas. When the meeting is over, the group is tired, so they decide to adjourn. They set up a new meeting time for when they will evaluate and prioritize all of the suggested features. The brainstorming ideas they have made so far are automatically saved. The facilitator retrieves the page with the list of ideas from the captured record and puts it on the agenda for the upcoming prioritization meeting.

### 2.3 Product Triage Meeting

This scenario shows a decision-making meeting, a contentious situation that requires structure to make any progress at all. The eNotePad supports this structure, allowing the facilitator to properly frame each person’s point of view with its supporting arguments.

Nine months after it commenced development on the latest version of Homework Helper, the product is in trouble. Overly optimistic planning and unexpected staff turnover have caused the development team to miss several milestones. Now, it is one week before the feature completion deadline, and it is clear that the company
will not make it. It holds a high-level meeting among the heads of the engineering, design, and marketing departments to decide what to do.

First, the facilitator asks the participants to enumerate their options. The engineering director suggests that the launch be pushed back by a month. The marketing director immediately contradicts that suggestion, but the facilitator puts the group back on track, reminding them that they will debate the issues later. The marketing director suggests shortening the time allocated to QA. The product manager suggests that they remove certain features from the product to make it launch on time.

Hearing no more suggestions, the facilitator now asks for arguments. The marketing director argues that the launch cannot be delayed, since there is an important trade show in six weeks. The product manager suggests that they can show a prototype only, but the marketing director replies that this would be suboptimal. Each point and counterpoint is written down by the facilitator, annotating the suggestion or argument that it supports.

The engineering director raises his hand and argues that the QA time cannot be cut. His point is that a buggy launch would be worse than a prototype, because of the bad press it would generate for the company. So, the team decides to cut features from the existing product to meet the feature completion deadline. The suggestion that was taken is annotated with a checkmark. The other suggestions are marked with an “X”, and they are minimized to create space to negotiate the specific features that will be cut. When the decisions are made and the meeting adjourns, the participants receive the decisions in their inboxes of their Object Browser. The department heads forward the meeting outcome to their groups, and they hold their own meetings to discuss what the new direction implies for their own work.

2.4 Military Mission Planning

The final scenario is a planning meeting to get a brigade of troops across a river and up a hill where they will camp for the night. It illustrates the advanced sketching features of the eNotePad. The mission commander projects a map of the area onto a
wall. The lieutenants use their Tablet PCs to mark the locations of their individual units. On the screen next to the map, the commander sets down the milestones of the deployment divided into six-hour intervals. For each time interval, the group decides where their units shall move. On the eNotePad, the commander sets up the current interval as a filter. The lieutenants copy their units and move them to where they would go during that interval. They further define that position to be an annotation of the previous position. If that position changes, the whole movement plan changes appropriately. When all is settled, the commander turns off the filter for that interval and moves to the next interval. Whenever the map gets too busy, the commander can hide some of the intermediate movement intervals by activating the filters that were set up before. Activating the filter of a particular time interval causes all of the new notes created for it to disappear. Deactivating the filter brings the notes back. If a particular advance needs to be revised, the lieutenant changes the position of the end-point. All of the dependent advances move with it. At the end, the team has a step-by-step plan for moving all of their troops to the new location with all of the plan’s dependencies built in.
Chapter 3

System Description

Having seen some of the ways we want the eFacilitator to be used, we will now describe the user interaction with our four programs in detail. The interaction enables the above scenarios and more in a flexible way.

3.1 Recommended Hardware Infrastructure

To function properly, the eFacilitator needs pen-based hardware. Ideally, every meeting participant has their own tablet PC. A meeting room doesn’t necessarily have to provide any support, though realistically, it should have at least one projector. The recommended configuration is four to eight projectors, depending on the size of the room. Projectors should be able to completely cover one long wall of the meeting room.

The content on these projectors can be controlled from a tablet computer. However, it is more natural for meeting facilitators to be able to write directly on the display surfaces. This can be done with electronic whiteboards such as the SMART-Board or the Xerox Liveboard. Alternatively, an existing whiteboard can be made touch sensitive with an eBeam or a Mimio receiver, which can detect strokes from special ultrasound-emitting markers. The latter devices are less reliable than the electronic boards, although they are significantly cheaper and more flexible.

Finally, a meeting room should have an audio and video capture capability. At the
high end would be a studio quality setup with multiple microphones and cameras and a technician to mix it all into one high quality media stream. For normal meetings, it suffices to have a microphone in the center of the table and a single, fixed position camera.

3.2 Object Browser

The Object Browser is a generic interface for finding and interacting with arbitrary objects. It combines some of the features of e-mail browsers, web browsers, and online libraries. The main window contains four panes: a toolbar at the top, a list modification dialog at the left side, and a right pane divided into two parts. The top part is the list itself and the bottom part is the view pane for viewing the details of any of the objects in the list. Note that this is the same layout as every modern e-mail browser. The similarity is apt, since e-mail is the most well developed structured, interactive object in use today.

The toolbar has standard buttons: a back button, a forward button, and a new button for creating new types of objects. It also contains all of the major operations that can be performed on the currently selected object. For example, if a meeting invitation is the currently selected object, the operations are “accept”, “tentatively
accept”, “decline”, “make comment”, and “request change of time or venue”.

The list modification dialog contains all of the ways the current list can be modified. The constraints that define the current list are visible, and any of them can be removed, widening the current list. Alternatively, the user can refine the current list by imposing additional constraints. To set parameters for a particular command or search constraint, the browser pops up a window that is itself another object browser, where she can use any object she can find to satisfy her parameter. For example, if she is browsing for people, there would be a constraint to let her focus on people belonging to a particular group. Clicking on this constraint brings up a new search window, beginning a search of groups.

The list dialog shows all of the objects satisfying the current list. The elements of the list are shown in column-delimited rows of data. The names of the columns depend on the type of data and the constraints used to generate it. Users can sort on any of the columns. Clicking on a row causes that element to become the current element, showing its operations in the toolbar, and putting its detailed view in the view pane.

The bottom right pane is the view pane, which is much like the main window of a web browser. Content can be in HTML or however the object wishes to render itself. Like web pages, an object’s view can have links to other objects. The links can navigate away from the current view or they can be in-line, expanding a bit of content without going away from the containing page. When a link is clicked, that object becomes the current object, and the user can manipulate it from the toolbar. Links can even be full-fledged queries, replacing the current list with a new one.

In the context of meeting capture and support, the Object Browser is used in four major circumstances:

1. Receiving announcements about meetings and responding to them.

2. In the meeting room, starting a meeting that has been scheduled or joining a meeting in progress.

3. Browsing for past meeting records and viewing a schedule of upcoming meetings.
4. Scheduling new meetings

Let us look at each of these in turn. The first two items are inbox-only scenarios. That is, the user does not need to browse anywhere to see these items. The inbox is a special folder of the Object Browser for receiving objects that an agent wants the user to see. Whenever someone schedules a meeting that our user is invited to, they receive the announcement in their inbox. As described previously, the actions they can perform are “accept”, “tentatively accept”, “decline”, “make comment”, or “request change of time or venue”. The announcement detailed view is the information view of the meeting. It shows where it is, when it is, who is invited, the status of the invitees, and all of the messages people have written about the meeting. Every major change in the meeting, such as a cancellation or a change of time or venue causes an announcement to be sent out.

After the user has seen the announcement, they can click on the special command “file away”, for inbox objects. This removes the object from the inbox. The user can find it again by searching on any its distinguishing properties such as sender or time.

Besides handling meeting announcements, the inbox is also the place where the user can interact with her immediate environment. In particular, if she enters a meeting room, the Object Browser’s inbox will receive all of the objects currently on the room’s channel. Her location can be determined by a location detecting device or by manual input. Thus, by walking into a meeting room, she receives all of the meetings that are ongoing or upcoming. Actions she could perform are to “start”, “join”, “stop”, or “restart” a meeting depending on its state. These functions are implemented by the Agenda Manager. They set up the room and connect her tablet computer to the meeting environment.

The last two scenarios, finding and scheduling meetings, are more complex than the previous two, because they involve browsing. To find a meeting, the user first searches for objects of type meeting. From there, they can search by date/time, location, the people or groups present, and the types of agenda items considered. Alternatively, they can browse general events by looking at the calendars of different people, groups, and locations. Calendars can only be searched by date/time, though
multiple calendars can be intersected together.

Users can now browse for meetings, but this requires them to also be able to browse for people, locations, and groups of people. To browse for people, a searcher can look at the people belonging to a certain group, the people housed at a certain location, and the people that another person knows. To browse for groups, the searcher can look at the groups a person belongs to, the groups housed at a particular location, and the groups that contain or are contained in another group. For example, I could look at all groups that are part of the AI Lab and which Howie is a member. To browse locations, a searcher can browse for locations of a certain type that are contained in or contain another location.

Note the recursive nature of browsing. A searcher could look for all people belonging to a group that contain a person belonging to another group, etc. This reflects the relational, non-hierarchical structure of social data.

When the user finds a meeting, the operations he can perform on it depend on whether the meeting is in the past or scheduled for the future. If the meeting is in the future, he can perform any of the functions described above. If he is the creator of the meeting, he can also edit the meeting or cancel it. If the meeting is in the past, he can only view it and any of the materials that were created or used during it. Viewing a past meeting brings up the NotePlayer, which can play back multiple views of the meeting in real time from any time point.

To schedule a meeting, the user clicks on the new button, selecting meeting as the object to create. She selects the time and the location. She creates a list of people and groups to invite. She can write a short subject and a longer meeting description.

They can also create an agenda. In the agenda, they add items to go in chronological order. The agenda items can have an owner and a type. Current possible types are a presentation, a review, a brainstorming session, or a decision meeting. A presentation can have an attachment of a slideshow. A review can include the documents under review. Agenda items can have any kinds of supplementary documents as well.

Note that to create a meeting, there must be people, locations, and groups in the
system. These objects can also be created from the Object Browser. A person is created with a name, various contacts like phone and e-mail, and various locations representing that person’s regular locations like home and work. The parameters to create a group are a name, an optional containing group, and then the list of people that make up the group. To create a location, the user provides a type, a name, and an optional list of containing locations.

3.3 The Agenda Manager

Now that we have seen how users can create and find meetings, let us look at what the eFacilitator does when the meeting actually takes place. A meeting has three states: upcoming, started, and stopped. The commands “Start,” “Stop,” and “Restart” shift between these states. In the meeting room, someone starts the meeting and the participants join it. Everyone who joins receives the Agenda Manager. The Agenda Manager has a list of all of the agenda items. There is always only one agenda item active at any particular time. For each agenda item, there is only one computer that is the controller. From the Agenda Manager, a user can start it, stop it, take control, or release control. Whoever starts the item is automatically in charge. For another user to take charge, the person who already has control must release it first. The controller’s computer affects what every other meeting participant sees. Every action the controller takes is reflected on the participants’ computers.

Every agenda item has a screen allocation: a recommended number of screens and a starting page for each screen. For example, a presentation could have one screen for the slides and one blank screen for developing more spontaneous content. If the room has fewer public screens than the recommended number, the screens have tabs to switch between the virtual screens. If the room has more public screens than recommended, the extras are used to show the most recent pages from the different virtual screens. For example, if a presentation has one virtual screen and the room has two screens, the extra one is used to show the slide just before the current one.

To avoid clutter, the pages shown on the public displays do not have any of the
page-switching tabs or toolbar functions that the tablet programs have. This feature can be changed if the public display is the only interface the controller has (e.g. if she is using an electronic whiteboard like a SMARTBoard).

From the perspective of the meeting participants, the interaction with the tools does not change with the meeting type. Each participant has an eNotePad. Every time the controller brings up a new page, the participant receives it. They can annotate this page however they want, and they can switch back to any of the previous pages that have been introduced during the meeting. If the controller changes the main page while a meeting participant is actively working on something, the participant will not receive the page change until they have been idle for five seconds.

After the meeting has concluded, the meeting leader clicks “Stop” on the Agenda Manager. The eNotePad record for each tablet is saved to a publicly accessible area. Each of these records is combined with the audio recording of the meeting. Later, when a user is browsing for a meeting, they can review the public view of the meeting, or they can review their own private view. Anyone who was not there can only review the public view. The public view of a meeting is equivalent to a meeting participant who doesn’t do anything, only listening to the controller’s actions and the public actions of the participants.

3.4 The eNotePad

The eNotePad is the core of the meeting capture application. It helps the users display and manipulate the ideas under discussion, serving as the group and the individuals’ working memory.

3.4.1 Previous Versions

The eNotePad is our critical note capture application, so it is a very challenging interface to design. We have made several iterations so far. The first version ordered notes in a vertical outline. Items could be expanded and contracted. Users could add notes under any item that was expanded. This proved to be limiting.
Next year's workshop

Location
Florida
Boston

Organizer
George
Sue

Figure 3-2: First eNotePad prototype: Users create an outline of ideas

Next year's workshop

Location
Florida
Boston

Organizer
George
Sue

Figure 3-3: The latest eNotePad: Users can move notes around and arrange them in a hierarchy
In the next version, users still had to create notes in a particular area of the screen, but they could move them anywhere on the screen. However, being required to write at a particular area of the screen to create a new note was also too limiting. The most recent edition lets users write anywhere on the screen. After a second of inactivity, the system computes the bounding box for everything just written and creates a note for it. These notes can then be moved around and arranged into any hierarchy.

After informal observations of people using this system, we have realized that the bounding box is too large to define the area of a note. Users like to draw large strokes to divide the screen, they often want certain types of notes to intersect each other, and they also like to draw arrows to connect things. These observations have driven us to the conclusion that a note-taking program must act just like paper for users to accept it. Users should be able to make any kind of stroke anywhere on the page, and that should not affect the notes that are already written down. A note-taking system should enhance the features of paper, but it should not remove any of the latter’s functionality. We are currently developing the latest eNotePad, which has the features and design philosophies that we describe below.

3.4.2 Interface Design Principles for Tablet PC’s

Before we go into the details of specific features, let us state a few principles dictating human-computer interaction on tablet computers in collaborative environments. First, users need to be able to operate the functions of the program quickly, since they are communicating with other people. The toolbar should be simple and commonly functions should have large buttons that users can click quickly.

To further increase user speed, we try to minimize the distance the user has to move the stylus. So, we place the menus to manipulate objects next to those objects and not on the toolbar or in the main menus. Also, we do not have a right mouse button. Users can only convey their intentions with pen gestures. Previous pen-based applications have solved the interaction problem by defining various arbitrary strokes correspond to different functions. We shy away from this approach if the gesture has no clear relation to the function being requested. We use a small number of familiar
gestures, clicks, double-clicks, strike outs, etc. If a complex operation needs to be arranged, we put it in a context-sensitive menu with instructive text or graphical labels.

We follow other principles specific to note taking applications that we have learned from our past experience. First, the user should be able to make any kind of sketch anywhere on the drawing palette. That is, there shouldn’t be any gestures that operate on open space. Previous editions of the eNotePad had implemented an auto-layout feature to keep sketches from overlapping each other. Users found this feature very frustrating, as they tend to do when the computer does something they don’t anticipate (e.g. Microsoft Office’s surprise paperclip).

3.4.3 High Level Goals: Organize, Associate, Annotate

The starting point for our new eNotePad feature set is a regular, paper notebook. The extra functions we introduce should interfere minimally with the paper functionality.

Most note-taking applications stop at this step, making no attempt to understand the segmentation or organization of what is written. However, we want to understand what humans are actually doing when they make marks on paper, and we want to aid them at this higher level. Through informal observation of people as they take notes or lead meetings, we have focused on enhancing paper by providing the tasks of organization, association, and annotation.

As we have mentioned before, visual work areas like notebooks and whiteboards are very important in aiding human problem solving and decision making. The visual aids serve as an external short term memory. They allow humans to comprehend complex, symbolic, structural relationships that would otherwise be impossible for them to process as a whole.

Organization, annotation, and association are the three ways we have observed that humans build up complex concepts through a visual manipulation of simple base concepts. Base concepts are simple, representational sketches that run the semiotic spectrum from realistic depictions to glyphs to total abstractions. For example, consider a tree. A human could represent this with a line drawing of a tree, a simpler,
Figure 3-4: Three types of visual representations people use to convey meaning

but still recognizable glyph. Or, they could use a word, whose relationship with the concept is only established through extensive training and discourse.

Organization is the process of categorizing concepts by grouping them together close to each other. Other things people do is draw boxes or circles around related items. They also draw tables or other kinds of graphs to divide up a page. The eNotePad lets users do all of these things to group and segment their notes. Notes can be moved anywhere on the page once they are created. When notes are in a group, the whole group can be moved as one. When the notes are in the cells of a table sketch, the table can be moved and its cells can be resized without losing the structure.

Association is the process of loosely relating items are not intrinsically related to each other. Humans tend to represent this visually with a linear icon like an arrow or an equals sign. In the eNotePad, we allow an association drawing between any notes. This association sketch stays connected to the endpoints even if they move around.

Annotation is the organizing of concepts hierarchically. That is, some collection of items may all be in reference to some main item. One concept can be a consequence of another concept, or it could be a response to it. This hierarchy is often represented visually as a list outline with the annotations indented under the main concept. In the eNotePad, users can place their notes into a hierarchy, where one note is an annotation of another. The hierarchy can be moved as one group. Also, it can be expanded or contracted, so that the annotations of a particular note are made visible.
or hidden, respectively. Furthermore, the eNotePad supports handwritten markup of lecture slides, web pages, or other kinds of documents, so that users' thoughts on the topics discussed can be written directly on top of them.

We believe that these visual relational operations are some of the key enablers of human creativity. Anecdotes from meeting facilitators and teachers bear this out. For brainstorming meetings, the standard facilitation trick has been to write ideas from the audience on different colored sticky notes and to arrange them in ways that foment further creativity from the group. From our own personal experience, when we design something or work out a problem, we write down all of our random ideas in a notebook, and solutions seem to magically emerge from the initially random clutter.

### 3.4.4 eNotePad Detailed Features

Now, let us look at all of the eNotePad's functionality in detail. The application is divided into tab-separated pages. Each page can start out blank, or it can have a background element such as an HTML page or a page from a PDF file. If the page has links, these can be clicked to change the page. The page may have implicit links, such as the slides of a slide-show, which implicitly link to the previous and next pages. These links appear in the toolbar if the current eNotePad is the presentation controller. The user can create new pages from the toolbar. These are initially invisible to the rest of the group until he chooses to make them public.

On the eNotePad drawing palette, the user creates notes, each of which is associated with a sketch. A note is the primitive sketch which is operated on by the organization, annotation, association methods. A sketch is made of many strokes from the pen. Each stroke has vertices or cusps, where the user slows down the pen and makes a sharp change of direction. We call the portion of a stroke between two cusps a smooth stroke. Stroke self-intersections or intersections with other strokes further subdivide the smooth strokes into contiguous smooth strokes.

To delete something from a sketch that they are drawing, the user can use the eraser. Alternatively, they can click anywhere on the sketch. This highlights the contiguous smooth stroke that was clicked on. The user can then draw a line through
that piece, and it disappears from the sketch. Double-clicking highlights the whole smooth stroke, which can also be deleted. Triple-clicking highlights the whole stroke.

When the user finishes drawing their note, they can click on an empty space on the toolbar, or they can wait five seconds. The computer takes what they have drawn and considers it as a completed sketch. The completed note can be dragged anywhere. The user selects a note by clicking anywhere near a stroke of the sketch.

Notes are initially private. The user shares them with the group by choosing "make public" from the context menu. This sends the drawing, along with all of its annotations to the rest of the room. Subsequent actions on this drawing are then distributed throughout the room.

Clicking on a note highlights it and brings up its context menu, to the right of its right-most points. Clicking on the note also creates a dashed bounding box around it. The user can drag this boundary line anywhere to create more space with the notes around it. The user can also drag the strokes of the note itself, growing or shrinking it. The surrounding notes move or expand relative to this resizing. Double-clicking on a note lets the user continue drawing the sketch, adding or deleting further strokes.

The context menu contains a variety of functions. There are seven major ones and four minor ones. The major ones are "annotate", "change parent", "expand/contract", "delete", "minimize", "copy", and "set/unset arrow". The major tasks have their own buttons that appear to the right of the sketch when it is clicked. The four minor functions are "make public", "organize group", "set/unset filter", and "edit filter". Figure 3.4.4 shows what this context menu looks like.

If the segmentation is not correct, notes can be split, and they can be merged. Each function is a separate button on the toolbar. To split notes, the user clicks the "split" button, and then they draw a stroke on the page. This stroke breaks up some of the notes. Whatever separate pieces the stroke creates become notes in their own right. For example, if there are two words on top of each other in one note, we can split this note into two notes by drawing a line between them (see figure 3.4.4). Merging combines separate notes into one. The user clicks on the merge button and they draw a stroke. Whichever notes intersect this stroke are combined together.
Figure 3-5: The eNotePad's context menu

Figure 3-6: An example of the splitting operation
Any user actions can be undone and redone with the undo/redo buttons respectively. The delete function deletes a note, together with its annotations. Copying a note puts it in the system clipboard together with its annotations. To paste a note, the user makes the right-click gesture, pressing the pen down and holding it in place for a second. This gesture opens the palette context menu, which has the paste function.

Annotations

Annotation is one of the system’s key capabilities. To annotate a note, users click on the annotate button. Subsequent notes are this note’s annotations. Annotations can go to any depth. When she is finished annotating, the user clicks on the annotate button again to get back to the main note’s level.

A drawing’s annotations are hidden and displayed with the contract/expand button. For finer control, any particular note can be hidden with the minimize button. Hiding a note hides all of its children as well. The change parent function lets a note set the parent note that it is annotating. This feature is used commonly in a brainstorming session, where ideas are haphazardly written down and subsequently categorized.

Related to annotation is the notion of grouping. Any note whose area contains another becomes the other note’s parent. When the user circles a group of drawings, all of them become children of the circle note. The user can move this circle around, and everything inside moves with it.

Users can also combine a set of drawings into a group without a circle sketch. The user selects the “organize group” function in the context menu of a note and circles the drawings she wants grouped. The note which did the organization becomes the head of the group. When the user moves the head around, all of the notes in the group move with it.

A final feature related to annotations is the notion of filters. A filter is a note that controls the visibility of another set of notes. When the filter is set, these drawings are not visible, and when the filter is removed, the notes are displayable again. A note is
automatically a filter of its annotations. However, while any note can only annotate one parent, a note can be filtered by more than one filter parent. To make one drawing a filter of another, the user clicks on the filtering drawing, and chooses “edit filter” from the context menu. Then, they can draw a line to any of the visible drawings to add them to the filter set. A note can be removed from the filter set by drawing a line through the line connecting the note to the filter parent. The user can also choose “set filter”, which makes the current note become a filter for subsequent notes until the user again selects “unset filter”. Filters are useful for complex annotation tasks like mapping, where a core drawing is marked up in several different ways.

Arrows

Arrows accomplish the association task. To make a note an arrow, the user chooses “set arrow” from the context menu. The system will attempt to find the drawings the arrow connects by finding the longest stroke and tracing it forward and backward. The user can manually set the source and destination using “change parent”. Since arrows can criss-cross the screen and get in the way of intermediary notes, the portions of arrows that intersect the boundaries of other drawings are not shown. Enough of the arrow should still be visible, so the user can see which two things it connects. Arrows are regular notes, so they too can be annotated with other notes. Once an arrow has been set, it cannot be moved.

Tables

To divide up a page or a portion of a page, a note taker can draw a table. A table is a regular note, and it is a useful way of organizing smaller notes, similar to a circle. The contiguous areas of a table are its cells. Each cell is a separate group. If a note is placed in a cell, the system keeps it there if the table moves around or part of it is resized. The strokes of a table are its rows and columns. As we mentioned, these can be moved once the table is drawn. Moving the dividing strokes expands the cells that are connected to this stroke in a logical way, similar to the resizing of tables in desktop drawing applications. So, tables are a useful organizational construct that
our system supports naturally.

3.5 The NotePlayer

The NotePlayer is our final application. It is used for browsing a meeting after it has taken place. When the user wants to find a particular discussion, he first uses the Object Browser, finding a list of past meetings organized by time, place, or working group. He views an individual meeting record with the NotePlayer. This record consists of whatever audio/video streams the room could capture, along with a timestamped list of actions on the eNotePad. The NotePlayer starts an eNotePad window and plays back the development of the meeting in real time while playing the audio/video record from the same time. Every change to the eNotePad is reversible via undo, so the user can move backward as well. Also, he can click on any note, and the NotePlayer will go to the time point when that note was created. The NotePlayer has a simple interface of a play/pause button and a large scroll bar for quickly scanning through the note and page progression.

Using the notes that are written down as an index into what is going on, the user can find the part of the meeting that interests them, and then play back the recording of exactly what was said. This player, combined with the Object Browser, is our method for supplying meeting retrieval.
Chapter 4

Implementation

Let us now look at the implementation details that enable the meeting capture automation and the geometrical operations of the eNotePad.

4.1 Scheduling and Finding Meetings

The objects that users find in the Object Browser are all agents: lightweight pieces of code that move from computer to computer and dynamically locate other agents to interact with. Agents serve as the interface to the devices in rooms and the mobile devices of individuals. Meeting agents use their knowledge of the meeting to operate the agents of the room and the agents on the meeting members’ computers.

This dynamic service finding is accomplished with a flexible query of a global, object-oriented data space. The agents locate the objects to interact with not by using a pre-defined address, but by querying the data space for objects that satisfy a particular set of properties. Human users make the same kinds of queries when they search the Object Browser. The data space is the key infrastructure behind all of our programs.
4.1.1 The Object Server

A simple, yet critical component of the system is an object catalog. This is like a web server for Remote Java objects. When an object is inserted into the catalog, it is given a unique key, comprised of its type and a unique integer. Every shared object in the world has a unique URL comprised of its catalog server and its key. Objects from one catalog can talk to objects from another catalog. Basically, the catalog is just an object location and identification service.

The catalog can be used to hold references to anything in the real world: people, places, things, devices, documents, e-mails, meetings. Generally, the objects stored in a catalog are small and lightweight. They only contain an object’s intrinsic properties and operations: the name of a person, the location of a room, the text of an email, the toggling of a light switch, etc. Individual objects do not contain their relations to other objects if this relation is not intrinsic to the type of the object. To find these relations, we use a semantic, object-oriented database.

4.1.2 The Object-Oriented Database

The object-oriented database stores heterogeneous relations among any objects in the world. A relation is a tuple of datapoints. A datapoint is a reference to an object or a constant like a number or a string. An example relation is ("located" Bob e21), where "located" is a string, and Bob and e21 are the URLs of the respective person and room.

To query the database, we create a pattern, and the database returns the set of relations that match the pattern. A pattern is a list of relation templates. There are two types of relation templates.

1. A constraint template puts a constraint on each the datapoints of a matched relation. A constraint is a set of possible datapoints and/or a variable identifier. An example is (office U 811), where U is the universal set of all datapoints.

2. An item template contains a single constraint and matches any relation which
has a datapoint that matches the constraint. \{Gary\} is an example datapoint set for all relations that contain Gary.

A variable identifier in a relation template list means that the points of a matched relation list which correspond to the same identifier must be the same. So, ((office ?1 U), \{?1\}) would be matched by ((office Gary 811), (captain "softball team" Gary)), but not by ((office Max 811), (captain "softball team" Gary)).

Datapoints are compared using a universal comparator. This total order acts in the expected way for objects with a standard ordering, such as numbers and strings. Vectors are compared by the number of their dimensions and coordinate by coordinate when their dimensions are equal. For example, \((1) < (0, 1) < (0, 2) < (0, 1, 2)\).

The variables in the templates allow us to find objects that satisfy any diverse set of relations. Here are some sample queries:

1. Asking for all people who have offices on the 8th or 9th floor with windows:

   (type ?1 person)
   (office ?1 ?2)
   (type ?2 room)
   (located ?2 "200 Technology Square")
   (floor ?2 [[8], [9]])
   (haswindows ?2)

2. Asking for all information about people at the AI Lab who play hockey:

   (type ?1 person)
   (member ?1 "AI Lab")
   (plays ?1 hockey)
   {?1}
The advantage of this database over traditional relational databases is its flexibility. The relations that satisfy the above queries could have been added by different agents, each without knowledge of the other. An agent can assert any kind of fact it wants, and another agent can query for any set of facts it wants. This contrasts relational databases, where the database designer must know exactly what kind of data is going into the database beforehand.

Relational databases have the advantage of being very fast on large data-sets. However, the kinds of data we store, random facts about people and places, tend to be broad and shallow. Also, for classes of relations that can become numerous, proper indexing can make lookups fast, as we shall see shortly.

**Solving Queries**

Queries like the one above are solved by combining indexes, each of which solve one particular part of the query. An index consists of a relation template list and a search tree of the relation lists that satisfy the template in sorted order. For example, an index that stored all offices in 200 Technology Square would have the template:

```
(type ?2 room)
(located ?2 "200 Technology Square")
```

To compare two relation lists \( r_1 \) and \( r_2 \) that match an index, each index has a term order. The compare function looks at the terms of \( r_1 \) and \( r_2 \) in this order until it finds two terms that are different. Datapoints are compared using the universal comparator.

For item templates, the comparison terms are the item that matches the template and the relation as a whole. For example, consider the index `((office ?1 811), {?1})`, all information about people who are in office 811. If we have the two relation lists, `((office Max 811), (likes Max pizza))` and `((office Gary 811), (vegetarian Gary))`, the comparison terms would be `((office, Max, 811, Max, (likes Max pizza)))` and `((office, Gary, 811, Gary, (vegetarian Gary)))`. The term order for this index would specify an ordering of these five coordinates.
Given a query, we find the best combination of indexes that satisfy a subset of the query's relation templates. We decide the best combination from a variety of heuristics, such as the number and type of query templates matched and the number of variables that would be bound. We need combinations, because sometimes a particular query subset cannot be handled by one index alone. For example, this happens if our query is `((type ?1 person), (office ?1 [800-999]), (likes ?1 ?2))`, and we have two indexes `((type ?1 person), (office ?1 [800-900]))` and `((type ?1 person), (office ?1 [900-1000]))`.

As a last resort, all databases have a universal index, which has the universal template as its template list. This index stores every relation in the database in sorted order. If no specific indexes match a particular query, this index is used.

When a record set returns from an index query, each record is resolved against the variable identifier, and then the query is run again on all of the relation templates that were not resolved. The process repeats until every relation template is resolved. Let us run through two simple examples to see how this works.

**Query Solution Examples**

We will solve the following query asking for people in the AIRE group whose offices have windows:

```
(member ?1 "AIRE")

(office ?1 ?2)

(haswindows ?2)
```

We will use the following indexes:

1. Universal Index

2. Triples index (U U U) with term order (1, 3, 2)

3. 8th-floor index:

```
(office ?1 [800-899]) {?1}
```
4. (office ?1 ?2)

First, using index 2, we resolve the first relation template, and we get all of the members of AIRE: Stephen, Kevin, Kimberle, Gary, Max, and Tyler. Then, using the universal index, we find the offices for each of these people: respectively, 832, 832, 805, 811, 811, 800. Finally, we check the last query using the universal index. Only 805 and 811 match it. Therefore, we return \{(Kimberle 805), (Gary 811), (Max 811)\} as our result set.

For our second example, we ask what Gary likes:

(likes Gary ?1)

We could use the universal index to find this quickly. However, assume we only had indexes 3 and 4. How could we solve this query? Whenever we can match a portion of an index and connect that portion to the rest of the index by a finite set, we can test the non-matched portion by substituting each possible value. If each value can be found, then this index can solve our query. So, [Gary] is a finite set. If we can find (office Gary [800-899]), then we can solve our query using index 4. We check (office Gary [800-899]) using index 4 to get (office Gary 811). Then, using index 3, we solve the query to find ((office Gary 811), (likes Gary baseball)). In general, making a pre-query to determine if we can use an index is inefficient. We only do it if there are no other indexes that work. With local databases, we have the universal index, so this never happens. However, in the distributed setting, it is sometimes necessary to do pre-queries.

**Database Set Theory**

All of the major database operations ultimately reduce to operations on sets. A relation template and a relation template list are each different types of cross product sets over U. A relation and a relation list can be represented as tuples of elements of U. We use this set theory to combine indices to solve part of a query.

The sets that define the constraint on a term of a relation template are datapoint sets. A *datapoint set* is a finite list of intervals in datapoint space ordered by the
universal comparator, \([a_1, b_1] \cup [a_2, b_2] \cup \ldots \cup (a_m, b_m)\). Intervals can be open or closed at either boundary.

Datapoint sets can be intersected, unioned, and subtracted from each other. Also, every datapoint set can return the least element greater than or equal to any particular datapoint \(x\). We call this value the least element above, \(\text{LEA}(A, x)\).

To make this work with open intervals, we introduce differentials and infinities. So, the open interval \((a, b)\) is actually represented internally as \([a + d, b - d]\) and \(\text{LEA}((a, b), a) = a + d\). The \(\cup\) set is represented as \([-\infty, \infty]\). The universal comparator knows how to deal with these new datapoints. \(\text{compare}(x + kd, y) = \text{compare}(x, y)\) if \(x \neq y\) and \(\text{compare}(x + kd, x) = \text{sign}(k)\). \(\text{compare}(\infty, x) = -1\). \(\text{compare}(\infty, x) = 1\).

The database clients have no knowledge of these special datapoints, so there is no way it could create a relation or query that referred to them explicitly. These special datapoints are very useful for navigating to the right place in a binary search tree, as we shall see.

Let us see how a query is solved by an index. The index stores all of the relation lists that match it as a search tree of \(n\)-tuples. For example, if we have an index \(((\text{office} ?1 811), \{?1\})\), the element stored is \(\text{office} \times \text{Gary} \times 811 \times \text{Gary} \times (\text{likes} \ \text{Gary} \ \text{baseball})\). The query is converted to an \(n\)-dimensional query set. Query sets and search trees can both implement the \(\text{LEA}\) operation for any point.

We want to find the points that are in both the search tree and the query set. To do this, we alternately move upwards in both sets until we find a point that is in both of them. Figure 4-1 illustrates how this works. Algorithm 1 shows the steps. \text{IndexTree} is the search tree and \(S\) is the query set.

The way a query set is made depends on the relation templates of the index that it is being matched to. Let us look at an example. Consider again our query \(((\text{likes} \ ?1 \ ?2), (\text{office} \ ?1 \ 811))\) with the index \(((\text{office} \ ?1 \ 811), \{?1\})\). Elements of this index are 5-tuples. The set representing this query is the 7-tuple \(\text{office} \times U \times 811 \times U \times \{\text{likes}\} \times U \times U\) with the equality constraint that terms 2, 4, and 6

49
Algorithm 1 Algorithm for querying an index

QueryBound = LEA(S, -∞)

NextIndexElement = LEA(IndexTree, QueryBound)

ResultSet = ∅

while NextIndexElement ≠ null do

while QueryBound ≠ NextIndexElement do

if QueryBound > NextIndexElement then

NextIndexElement = LEA(IndexTree, QueryBound)

else

QueryBound = LEA(S, NextIndexElement)

end if

end while

ResultSet = ResultSet ∪ {NextIndexElement}

NextIndexElement = FollowingElement(IndexTree, NextIndexElement)

end while

---

Figure 4-1: Finding the points of an index that match a query set
must be equal. When we do the above algorithm, we must convert an element of the index set to this 7-tuple, find the LEA for that point, and then convert it back to a 5-tuple.

The two kinds of sets we manipulate are cross sets with equality constraints and cross set unions, which are finite unions of the former. We use cross sets for finding the elements of an index that match a query. Conversely, to find a combination of indexes that will contain a query, we use cross set unions. Cross set unions are powerful, because they are closed under union, intersection, and subtraction.

Our index querying algorithm can have fast or slow performance depending on how well the indexes anticipate the queries asked of them. For example, if I have the index (likes ?1 ?2) with term ordering (1, 2, 3), and I ask for all relations of the form (likes Howie ?1), I will get all of the people that Howie likes very quickly, because they will be adjacent to each other in the index. However, if I ask (likes ?1 Howie), it could be very slow, because I will need to examine every person who likes someone to see if they like Howie.

Thus, it is sometimes important to keep indexes for several different orderings of terms in a relation template. In our example above, with the universal index, we had two orders for the triple relation template (?1 ?2 ?3): (1, 2, 3) and (1, 3, 2).

Triggers

How does the database know how to insert a particular relation list into an index? When a new relation is added, we look at every index. If there is a match on a template of an index, we query the database with the rest of the index bound to the datapoints defined by the new relation. For example, if our index is ((located ?1 ?2), (type ?1 room)), when we add the relation (located 811 "200 Technology Square"), we query for (type 811 room). If we find this relation, we add the pair to the index.

For every relation, the database maintains the list of indexes that store that relation. So, when the relation is removed, the database can know which indexes to remove it from.
The mechanism for notifying indexes of the addition and removal of relation lists can be applied to any list of relation templates. In this way, database clients can add triggers. The client receives a callback whenever a particular relation combination that fits the trigger template is added or removed. Triggers are the mechanism by which agents can monitor a database for interesting events.

4.1.3 The Semantic Database

The above system describes an object-oriented database. It is good at storing and retrieving all sorts of relations. However, it does not have any logic. To have a truly useful system, we need inference rules and truth maintenance. We will show how these rules can easily augment our database, making it much more powerful.

Inference Rules

The semantic database has support for forward-chaining inference. An inference rule is any sort of common-sense rule, such as \((\text{using} \ ?1 \ ?2), (\text{located} \ ?2 \ ?3)) \Rightarrow (\text{located} \ ?1 \ ?3)\), which states that if someone is using a certain device, then that person is in the same location as the device. Whenever we add a relation to the database, we check to see if any of the antecedents of any inference rules become true. If they do, we add their resulting relation, which in turn can trigger additional inference rules.

The bold relation, \((\text{using} \ ?1 \ ?2)\) is called the master relation. We need this relation, because the database supports contrapositives, \(A \Rightarrow B\) implies that \(\neg B \Rightarrow \neg A\). Hence, if some relation in the result of an inference rule is removed, some relation from the antecedent must be removed as well. The master relation is defined by the person who adds the rule to the database. An if-and-only-if rule \(A \iff B\) can be defined by creating two inference rules, \(A \Rightarrow B\) and \(B \Rightarrow A\).
Truth Maintenance

Truth maintenance is the way for the database administrator to ensure that relations that become false get purged. The contrapositive rule is one way to achieve this. The semantic database also can have uniqueness maintainers. A uniqueness maintainer ensures that at most one relation list out of a class of relation lists is true. For example, a uniqueness maintainer to make sure that someone can only be in one room at a time would be \((\text{located} \ #1 \ ?2) \ (\text{type} \ ?2 \ \text{room})\). \#1 is called the distinguishing variable and \((\text{located} \ #1 \ ?2)\) is the master relation. Assume we have the relations (located Max 811), (located Kimberle 835), (type 811 room), and (type 835 room). If we then add the relation (located Max 835), the master relation with the matching distinguishing variable, (located Max 811) would be removed.

For any uniqueness class, we can also set a default relation. For example, to our above uniqueness maintainer, we can add a default relation (located #1 unknown). With the default relation, if we remove (located Max 811), (located Max unknown) would be added in its place.

With uniqueness maintenance and inference rules, it is easy to define rules that contradict each other. A simple example is a rule like \(A \leftrightarrow B\) with a uniqueness maintainer of \(A \parallel B\). To avoid infinite loops, whenever we add or remove a relation to the semantic database, we first follow through all of the implications of the addition or removal. If no contradictions occur, we add and remove all of the relations that our rules have told us to.

4.1.4 The Distributed Semantic Database

The semantic database is useful for agents talking to each other on a single computer. However, we want agents all over the world to exchange information in one large, distributed data space. An agent needs to know where to send its knowledge, and it needs to know where it can go to look up different types of information. It is not scalable to put all of the relations in the world in a single database.

Fortunately, it is easy to set up a network topology and a database scope hierarchy
to make a large, interconnected web of data manageable. We organize databases in a directed, acyclic graph (DAG). Each database has a public scope, composed of scope elements. A scope element consists of a relation template list of the relations this database stores and is interested in. The scope element contains other routing information as well.

For a particular database, the databases "below" it on the graph are called the child databases. The databases above it are the parents. When two databases make a connection, each database combines the scope of all of its connections and sends it to the other. The parent/child relationship helps keep the extent of this scope combination reasonable, because a child database can consolidate the scopes of all of its children, letting the parent treat the child and all of its children as just one database. Also, when two databases make a connection, one examines the scope of the other and sends any relation lists that match the other's scope. This synchronization happens whenever a database changes its scope.

Every scope element has a consolidation level which is like a measure of how far a particular database is from the actual data relative to the current database. When a database wants to have every relation from a certain scope, it has a scope element with a consolidation level of 0. When we want a particular database to be a consolidator for its children's scope, we define a scope element whose relation template list encompasses its children and whose consolidation level is 1. When a database sends its scope to its children, it converts all of its scope elements of level 1 to level 2 and eliminates redundant scope elements. When the database sends scope to a parent, it converts its level 1 scope elements to 0 and reduces redundancy.

Some examples will clarify the situation. Let us say we have six databases in the configuration displayed in the figure. Databases 1, 2, and 3 have the scopes (office ?1 [800-810]), (office ?1 [810-820]), and (office ?1 [820-830]), each with level 0. Database 4 is a scope consolidator for the 8th floor. It has a scope element of (office ?1 [800-899]) with level 1. Database 5 represents the 9th floor with a scope element (office ?1 [900-999]). It may or may not distribute this information among a set of children. Database 6 represents the whole building, and
Figure 4-2: Six databases in a hierarchy of scope

it has a scope element (office ?1 [0-999]).

These databases are connected as shown in the figure. Table 4.1 shows how each database understands the scope of the universe. Any of these databases can answer any question about offices, but each does it in a different way. When a database has a relation list that matches a particular scope element, it only sends it to the remote database if that relation list was not already sent to a scope element of a lower consolidation level. This prevents root databases from needlessly sending their relations to their scope consolidator.

When we query the distributed database, it first finds the scope elements with low consolidation level which intersect the query. If these elements do not cover the whole query, the database moves up to the higher level scope elements, which capture the leftovers. We do this until we have found scope elements to cover the entire query. When we send our query on to the database with the higher level, we tell it all of the lower level scope elements we used. This prevents a scope consolidator from trying to resolve the query by giving the exact same query to a scope element we have already used. In other words, we want the scope consolidator to only find the query matches
Table 4.1: How each database views the scope of the world

<table>
<thead>
<tr>
<th>Database</th>
<th>Scope Element</th>
<th>Source</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(office ?1 [800-810])</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [810-820])</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [820-830])</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [900-999])</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [0-999])</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>(office ?1 [800-810])</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [810-820])</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [820-830])</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [900-999])</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [0-999])</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>(office ?1 [800-810])</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [810-820])</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [820-830])</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [900-999])</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [0-999])</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>(office ?1 [800-810])</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [810-820])</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [820-830])</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [800-899])</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [900-999])</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [0-999])</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>(office ?1 [800-899])</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [990-999])</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [0-999])</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>(office ?1 [800-899])</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [900-999])</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 4.2: The scopes for our circularity example. An inserted relation will loop forever.

<table>
<thead>
<tr>
<th>Database</th>
<th>Scope Element</th>
<th>Source</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(office ?1 [900-999])</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [900-999])</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>(office ?1 [900-999])</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [900-999])</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>(office ?1 [900-999])</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [900-999])</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>(office ?1 [900-999])</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [900-999])</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [900-999])</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(office ?1 [900-999])</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

from the spillover that was not caught by our low-level consolidators.

The distributed database also sets up scope elements as triggers with the local database to listen for new relation lists that match the scope template. When a relation is added or removed, the proper triggers are fired, and the database sends the new relation list or deleted relation to the source of the scope element.

However, propagating a change like an insertion or deletion is actually more complicated than this. Consider the case where we have three databases, each replicating the same scope, (office ?1 [900-999]) with one parent. Chart 4.2 shows how all of the scope elements get resolved in this network. For queries, this is fine. However, when we try to insert a new element, (office Jack 910), we will loop forever, since every child believes that the parent is interested in the relation, and the parent believes that all of the children are interested.

To fix circularity problems like this, we augment the scope element to have a datapoint set called the address set. This set contains the addresses of all of the final databases which are interested in part of this scope. So, a consolidating scope element would combine the address sets of all of the scope elements it was consolidating. Here is how our 9th floor example looks with address sets. Whenever we insert or remove a relation remotely, we include an address set. When we find scope elements that match the relation, we split up their address sets intersected with this set. We forward the
insertion/removal to the remote databases with the new address sets.

In our example of the 9th floor, we first insert (office Jack 910) into database floor9 with address set U. The scope is resolved into floor9.1 through floor9.5, and 5 insertions are made. After inserting the relation locally, each database stops, since there is nowhere else for the relation to go.

The distributed, semantic database is infinitely scalable, and it is also arbitrarily robust, since there can be many children databases attached to many parent databases. However, care must be taken while setting it up, because every database does in fact theoretically contain the scope for the entire world. If this scope is not adequately partitioned beforehand, the databases on the network could be overwhelmed by having to work with a huge number of these scope elements. In practice, it is best to not have all of the databases in the world connected like this. Most facts are used very close to where they were created. So, to manage the complexity, we maintain different database networks for each of the different types of data that agents in the world will be likely to share. An example is a user’s personal information store, where they organize their e-mails, documents, calendar, and other miscellaneous items. Most of this data is all local to the user, so the database that stores it is read-only. It listens for certain kinds of facts, but it cannot be queried for most of them, and it doesn’t trigger other databases when new relations are added. When disparate items need to be shared, there can be special objects to do the bridging.

To see how a particular knowledge network could be set up, we will set up a network to locate a service anywhere in the world. Generally, there are two ways a physical resource locates itself: numerically and logically. Numerical coordinates are latitude and longitude. Logical coordinates are all of the things that are used in addresses: cities, states, zip codes, streets, buildings, floors, suites, offices, etc. Logical coordinates should be able to map to numerical coordinates, though the opposite is not true.

It is easy to set up a network for numerical coordinates. Each database in the network will represent some rectangle of latitude and longitude. The database at the top of the network will have a rectangle for all of earth. The ones just below it would
generally be the rectangles of countries. The ones below those would be rectangles for states, etc.

If we want to find all Chinese restaurants within five miles of our current location, we use a location detector like GPS to get our latitude and longitude l and d. Then, we do a query:

\[(\text{type } ?1 \text{ restaurant})\]
\[(\text{style } ?1 \text{ Chinese})\]
\[(\text{latitudelongitude } ?1 \ [l-5, l+5] \ [d-5, d+5])\]

Each rectangle would have a scope element of the form:

\[(\text{latitudelongitude } ?1 \ [l1, l2] \ [d1, d2])\]
\{?1\}

So, we would first do a query to get all restaurants within 5 miles. Then, for each one, we would ask if it’s style was Chinese.

Performing lookups from logical coordinates is very similar, though the scope consolidators correspond less to connected physical regions. For example, states are organized in alphabetical order, so a high-level consolidator would contain the index:

\[(\text{state } ?1 \ [\text{AL-MT}])\]
\{?1\}

Now, if we wanted to look up all the casinos in Massachusetts, we would perform the query:

\[(\text{type } ?1 \text{ casino})\]
\[(\text{state } ?1 \text{ MA})\]

This would be resolved the same way as the previous queries. There is a problem with both of these examples, however. The global scope consolidators are asking for all pieces of information for everything located in the world. This means, whenever we add a new relation, we have to query the location of each of the terms in the
relation to determine if we should forward the information on to the rest of the world. Performing several distributed, multi-hop queries for every database insertion can be quite time-consuming.

One possible solution is for every database to store the locations of every object in the world. So, the lookups required for inserting a new relation would be local and fast. However, this is not feasible, since there are too many objects in the world, and they are always moving around. Keeping all of the databases in the world up to date with these changes would not be possible.

The solution is to make every scope element have a local only flag. This means that to test if a relation is part of a scope element, we only query the rest of the scope element at the local database. For example, if I know something about a paperclip in Italy, I only share this information with Italy’s scope consolidator if I already know that the paperclip is in Italy at the local level. Any scope element which has an item template in it should be local only.

4.1.5 The Object Browser

The distributed, semantic database is a general-purpose tool that can be applied to many ubiquitous computing tasks. The user interface to the database is the Object Browser. Its architecture is quite general, making it suitable for browsing any list of objects. In today’s software, when the user performs some sort of networking task, they do it by manipulating a list of objects. In search engines, email programs, file-system browsers, news sites, shopping sites, and message boards, users manipulate lists of web pages, emails, documents, stories, products, and messages respectively.

However, each of these different types of browsing programs can only handle their own particular type of data. Also, users must learn a slightly different UI for each list browsing program, which can be a bother. Keeping with our philosophy that any two networked objects can be related to each other, any set of objects ought to be viewable in the same list. Also, they should be browsable on any device, from a computer to a PDA to an audio-only telephone. The Object Browser’s accomplishes this.

60
The browser always has two components: the current collection of objects and all of the ways the current collection can be modified. In some circumstances, the inbox also lets the user add a new item to the dataspace. The current collection corresponds to a query that the inbox has made over all the objects of the world. The collection modifications are all of the ways that the current query can be modified to generate a new collection.

For every collection, the objects returned fall into a variety of types. A type is a set of relations an object can satisfy. For each type, there is a list of properties that the objects of that type can satisfy. A property is the user's view of a database relation template. A property has a name and, recursively, a list of types that the values of the property must satisfy. So, the modifications to narrow the current collection are all of the properties of all of the types in the current collection. As we mentioned in the system description, the user fills in the values of the properties through another Object Browser window. A property can have a default sub-query associated with any particular value. So, when she fills in that value to refine a query, the default sub-query serves as the starting point. For example, when she chooses to invite a person to a meeting, the default query is the set of people the user has had meetings with recently. Besides narrowing the query, she can also expand the current query by removing any of the properties used to construct it.

Each object in a collection contains various human-readable descriptors of itself. It also contains all of the operations that a human can perform on it. Given the type of the collection, the inbox figures out the best descriptors to use in object list. For example, if the current collection was produced by asking for all AIRE group meetings where food was ordered, the descriptors would be the meeting type, the time, place, and the type of food. "AIRE group" is not a relevant descriptor, because it is implicit in the query. However, if I asked for all of the activities I had to do today and an AIRE group meeting was on the list, "AIRE group meeting" would be a descriptors. The food ordered would not be included, because it is not an identifying property under this query context. Objects in the Object Browser also know how to produce a detailed view of themselves. This is the content that is shown in the Object Browser's
view pane.

The Object Browser can be used in many ways, and we have certainly not explored all of them. Here, we will describe only four types of objects that are relevant to our task of finding and scheduling meetings: people, groups, places, and events. At the moment, users cannot create their own properties and types, so the ones we provide need to be comprehensive.

Browsing people is the most complex type, since there are so many ways people can be classified: gender, age, education, profession, where they live, where they work, what their hobbies are, what clothes they wear, what music they listen to, etc. We choose two properties: what groups they belong to and who they are known by. These are the most common ways people identify and locate each other when they are acquainted. And these are the most common properties a meeting organizer would use to find people to invite to a meeting. The properties used for searching people, groups, places, and events are summarized in table 1.

Meetings are one type of event the user can browse for. When he finds the meeting he is looking for, he can click on it and view all the details in the content pane. The details include the time and place and all of the people who plan to attend and those who were invited, but who declined or didn’t reply. It also includes a message log: the list of announcements and any comments the participants might have made about the scheduling or the agenda, etc. If the meeting has already taken place, it includes a link to the meeting record. From the record’s view pane, the user can access the final meeting output or any of the individual pages of it. He can also browse the meeting using the NotePlayer.

If the meeting has not yet occurred and the user’s current time and place are far away from the meeting’s scheduled time and place, he can perform several scheduling tasks on it. He can accept or decline the invitation, add himself to the list of participants, or request that the meeting be rescheduled or cancelled.

If his location is the location of the meeting and the current time is within an hour of the meeting’s scheduled time, he can start it. This starts the Agenda Manager. This program locks the resources of the room, and it prepares the environment. It
turns on the lights and places the meeting materials via the eNotePad on whatever public displays are available. It finds the interfaces to these devices using the database. When a meeting started, another user can join in. When she clicks “Join”, the Agenda Manager creates an eNotePad instance on her local machine, synchronized to all of the other eNotePads in the room.

Besides browsing and interacting with objects and events, the Object Browser also lets the user create them. An object’s type defines all of the properties that a new object can have. The type can define a custom creation interface, or the Object Browser can automatically generate this interface based on the property list.

4.1.6 Channels

The last component of the infrastructure to create and browse meetings is the content channeling system for bringing items of interest to the Object Browser’s inbox. A channel is just a place where content producers put their content. On the receiving end, channel listeners subscribe to all of the different channels they are interested in. Channels can be connected to each other, so that one channel can listen to another channel and publishes the latter’s content as its own.

At the interface level, the user has a set of channel listeners that continually bring items to her inbox. Once she has looked at a message, she can file it away, where it is indexed by date, author, and the chain of channels that led it to the user’s box.

Let us look at the channels for the meeting management application. Every meeting has an announcements channel. Everyone invited to the meeting is automatically added to this channel. Meeting reminders and creation, rescheduling, and cancellation announcements are sent over the channel with all the relevant information. The meeting itself is added to the channel of the room where it will take place. The room in turn has a channel which publishes all items that are taking place that day. When a mobile computer announces its presence the room, it automatically receives this channel. In this way, when a user enters a meeting room, she gets the ongoing or upcoming meetings in her inbox.
4.2 eNotePad Implementation: The Geometry of Handwritten Notes

The eNotePad is a highly interactive GUI program. As such, it uses a version of the model view controller architecture. This architecture stipulates that a program is divided into a number of simple functional units that are fully independent of each other. One unit is the application’s data model: a compact representation of all of the information gathered from the user. The model is the core of the program. To persist the program, this is the piece that gets serialized. Given only the data model from a point in time, the application can reconstruct itself. This structure is itself often an agglomeration of smaller data structures.

Another unit is the rendering engine. This component has its own data structure, the view structure. The renderer has everything it needs to display the application on the screen from this view structure. Like the data structure, the renderer is composed of smaller units, each of which is capable of rendering a certain small object like a window, button, or menu. The view structure is completely separate from the application structure. This way, the application’s graphics can be tested separately from the data structure.

To make the view reflect the data model, there is a controller. It listens to change events from the data structure, and it updates the view structure correspondingly. The relationship is shown in figure 4.2. For the eNotePad, we generalize this architecture, so that there is a multi-stage rendering pipeline with a data structure at each stage and controllers between the stages. The further down the pipeline we go, the closer we get to the actual pixels that are shown on the display.

Above the model, view, and controllers are a diverse set of command executors, each of which performs a specific application command. The commands can work at any level of the rendering chain. For example, resizing a window would happen further down the pipeline from entering a piece of data into a table. To actually enable the interaction, there are a variety of events and event listeners. Events are fired from input units. Various input units monitor what the user does in concert.
with the application data structures. When a particular input unit sees something interesting, it fires an event and all of the listeners are notified. The event listener's function could be to modify another input unit or it could be a command dispatcher that converts the event into the parameters of a command.

A diagram of all of these components working together is shown in Figure 4-4. Let us look at some of the important data structures for the eNotePad.

The eNotePad's core data model is a sequence of pages. Each page has a background element, which can be an HTML page, a page in a PDF file, or just an ordinary single-color background. The page's content is a collection of notes. Each note is associated with a particular sketch. The notes are organized into two hierarchies. The grouping hierarchy determines which notes belong together when they are moved around. The visibility hierarchy determines which notes can control the appearance of other notes.

4.2.1 Geometries

Geometries are our core objects for manipulating digital ink. A geometry is a union of points, line segments, and areas. It is initially constructed as a union of line segments on the plane. These segments naturally divide the plane into different regions, and the geometry can define any of these regions as filled or not. The lines also define a natural set of vertices, points which are connected to more than one line. Vertices do not necessarily have to be the end-points of the lines, since two lines may intersect.
Figure 4-4: Advanced Model View Controller Architecture
each other.

Given a set of lines from a user sketch, we want to find the regions and vertices that it defines. Finding the vertices is straightforward. We first pixellate the real plane and then find the line segments which intersect each pixel. For every pair of line segments in a pixel, we find their intersection. If it exists, we add it to our vertex list.

To find the regions formed by a set of lines, we first locate all of the vertices. Each vertex has a set of line segments emanating from it. Every segment emanating from a vertex has a region to its left and to its right. Equivalently, every directed segment has a region to its left. (The right side of the segment \((p,q)\) is the left side of the segment \((q,p)\)). Every region is to the left of some directed segment, so we just have to equate the left regions of each directed segment, and we will have found all of the regions in the geometry.

To identify all the regions, we make tours around the boundary of each region until we have identified each directed line segment with a region. To make this tour, we start at a segment \((p_1,p_2)\). At \(p_2\), we choose the connecting line segment closest to our left. An analogy is that we are driving in a car from \(p_1\) to \(p_2\). We come to an intersection at \(p_2\), and we choose the road closest to our left. This segment will be \((p_2,p_3)\). We continue this tour until we get back to \(p_1\). Each directed segment we visit is identified with the same region.

This algorithm shows that we can quickly find the regions of every set of line segments in the plane. This ability lets us perform standard set operations on geometries. To find the union of two geometries \(G_1\) and \(G_2\), we just combine the line segments that make up \(G_1\) and \(G_2\), find the resultant vertices, and fill the regions that are filled in either \(G_1\) or \(G_2\). Intersection is similar. We combine \(G_1\) and \(G_2\) and find all of the vertices. We remove all vertices that are not in both geometries. Then, we fill in the areas that are in \(G_1\) and \(G_2\).

Several other important geometry operations are the approximate expansion, the interior, and the convex hull. The expansion of a set of points \(P\) is the set of points \((x,y)\) which are within a distance \(r\) of a point of \(P\). For example, the expansion of
a point is a circle. Since the expansion of a geometry can have curved surfaces, we
cannot take the exact expansion. However, we can approximate the curved surfaces
with lines as accurately as we like, so that our expansion can be another geometry.
The interior of a geometry $G$ is $G$ with all of the interior regions filled in. For every
geometry, there is always one region which stretches to infinity on all sides. This is
called the outside of the geometry. The interior is all of the regions that are not the
outside region. The convex hull of a geometry is the convex hull of all of the vertices.
The algorithm for finding the convex hull of a set of points is standard and can be
done in near-linear time in the number of points.

4.2.2 Sketches

One level above a geometry is a sketch, what the user creates with the pen. A sketch
is a collection of strokes. A stroke is a continuous sequence of pen movements. It
is represented as a list of coordinates, found by repeatedly polling the pen location.
A stroke is split up into smooth strokes. The boundary between two smooth strokes
is a point where the user makes a sudden change of direction with the pen. These
points can be detected by finding the places where the pen slows down and where the
direction of movement changes dramatically.

Using the geometry of the sketch, we divide it into sketch segments. A sketch
segment is a portion of a smooth stroke, bounded by the smooth stroke end-points or
by an intersection with another stroke. So, a sketch has three levels of granularity:
strokes, smooth strokes, and sketch segments. As we described, either of these things
can be removed from a sketch by single, double, or triple clicking.

Sketches can also be merged with each other, and they can be split apart. Merging
is simple, constructing a sketch from the components of two or more sub-sketches.
Splitting, as described previously, is done when the user draws a stroke to delineate
the separation line between two parts of a sketch. Here are the steps:

1. Let $G$ be the geometry of a sketch $S$.

2. Let $G'$ be $G \cup \text{Polygon(ConvexHull}(G)) \cup \text{SplitStroke}$
3. Find the contiguous regions of $G'$. These are the regions that are connected to each other that do not cross the outside of $G'$ and do not cross the SplitStroke.

The contiguous blocks are the new sub-sketches. They are composed of the sketch segments from each block.

Finally, a sketch can find all of the sketch segments that are near any given point. This allows the stroke deletion process described in the system description. To do this, every sketch segment is expanded a small amount separately, and an entry is made in every pixel coordinate that the expansion overlaps. Then, when the system queries a particular mouse click location, the sketch can tell the system precisely which sketch segments the user intended to click on. The union of all of these expansions is the clickable area of the sketch. This is where the user clicks to show the object’s menu and to move the object around.

In addition to the clickable area, every sketch also has a bounding area, the area a sketch is considered to bound. For example, circles bound everything that is inside them. We define the bounding area of a sketch to be the interior of an expansion of the sketch. We expand the sketch, because we assume that the user is sloppy in drawing their circles. The expansion corrects any gaps between the starting and ending point. Users are less accurate when they are connecting points of a circle than when they are attempting to click on a particular area of the screen. Therefore, the expansion that defines the bounding area is larger than the one that defines the clickable area by about a factor of 2.

We have verified this increase in inaccuracy through informal experiments. It is also a consequence of Fitts' Law of psychomotor performance, which states that the targeting accuracy of a human appendage decreases with the speed of that appendage. When we draw a circle while we are taking notes, we typically do not slow down to connect the points, so our error is large. However, when we click at some point, we are not moving, so our accuracy is greater.
4.2.3 Notes

Now that we have the primitives for manipulating general sketches, let us look at the operations for manipulating arrows and shrinking and stretching notes.

Arrows

Arrows are a drawing to show the relationship between two other drawings. An arrow can have any sketch associated with it, though we expect it to be roughly one-dimensional, in the general shape of some type of arrow. An arrow is always associated with its source and destination drawing. The unique capability of arrows is their ability to change direction and shape depending on the position of the two parents relative to each other.

When we have an arrow drawn from object $X$ to object $Y$, it should still visually connect $X$ and $Y$ even when $Y$’s orientation relative to $X$ changes. To do this, we consider the arrow to be defined relative to the mid-line between $X$ and $Y$. The mid-line is the line segment connecting the centers of $X$ and $Y$. We consider the points of the arrow sketch relative to the mid-line outside of $X$ and $Y$ and the unit normal to this external mid-line. As the external mid-line changes in length and direction, the arrow sketch changes relative to it.

The system determines the arrow’s source and destination by examining the longest stroke of the arrow’s sketch. The system follows the tangent of the beginning of this stroke backwards, and labels the first object this tangent hits as the arrow source. It follows the tangent of the end of the stroke forward to find the destination object. The “change parent” functionality lets the user change the source and the destination if the system gets it wrong.

4.2.4 The Shared eNotePad

Now, we have seen how all of the pieces of an individual eNotePad work together. How do we synchronize many eNotePads together, so that everyone in a meeting has the same view of the discussion, that the participants are literally on the same page?
The mechanism is distributed commands sent through the broadcaster object. All of the eNotePads within a meeting have a link to this object. The broadcaster maintains a thread list of command listeners. Whenever it receives a command, it executes it in parallel on all of its listeners. Furthermore, the broadcaster contains a local data model, which receives all of the commands it gets. When a new user joins the meeting, the meeting manager sends the user a seed for the eNotePad as well as the broadcaster. The user's computer uses the seed to start the eNotePad, and it connects to the broadcaster, getting the latest data model.

Commands that simply manipulate one notepad's view of the data are not distributed. However, when a user does something fundamental to the drawings in the grouping or visibility hierarchies, such as adding a page, making a note public, deleting a public note, moving it, or changing its parent, it is broadcasted to everyone. There are specialized commands that do synchronize the views of all of the objects, but these must be specifically invoked by the user from the toolbar.

To filter the kind of commands it gets, any program can set up a routing policy with the broadcaster. It only sends commands to a receiver if the program and receiver both agree that the receiver should get that type of command. For example, if someone is giving a talk with slides, all of the users are making notes and sharing them on their eNotePads. The particular eNotePad that is showing the slides should not be accepting the commands to add these notes, since that would interfere with the presentation. This routing policy can also be used to send private notes to one or two others in the meeting.

4.2.5 The NotePlayer

We have already seen how the record of commands on the eNotePad is stored. The Meeting Manager couples this record with the audio record, a standard, sampled audio waveform. The NotePlayer couples these two records. The transactional properties of the eNotePad commands let the NotePlayer go to any point in the meeting. Going to this point in the audio is very simple. So, the implementation of the NotePlayer is very straightforward, combining the eNotePad's display component with audio playback.
generated from Java Sound.
Chapter 5

Previous Work

Meeting support and capture has been one of the most active areas of ubiquitous computing over the past decade. The seminal work has been done at Xerox PARC and at the Georgia Tech Classroom 2000.

5.1 Evolution of Electronic Conference Rooms

As personal computers began proliferating in the late 80's, several research groups put them in conference rooms. Some examples are the CoLab at PARC [27], the Capture Lab at the EDS Center for Machine Intelligence [17], and the Collaborative Management Room at the University of Arizona [22]. These rooms all featured large, projection displays. The meeting leader would interact with this display through their own computer at a podium or a desk. The participants would summarize their ideas on the local PC's, and they would put them up on the large screen.

Even though the meeting record could be captured quite accurately, the setting of people looking at and typing into their computers instead of looking at and talking to each other proved to be odd and unnatural. User studies showed that there was much more information exchanged in face-to-face meetings than simply the words that were used. In particular, one study of user interface designers showed that 40% of the visual actions were actual marks on the page, while the other 60% were transient hand gestures about these marks [2]. When the same designers had the same kind of
meeting remotely, they had to write down nearly three times as many things to get
the point across.

In response to these and other problems, Weiser defined the goal of ubiquitous
computing to be making computers invisible. People should be able to interact with
each other without even realizing computers are present. Following this mandate,
PARC developed the Liveboard [8], an interactive, electronic whiteboard. People
could sketch on this board just like a regular whiteboard, and they could immediately
gesture at it in the way they were familiar.

Abowd et al. extended the ubicomp vision from the corporate environment to the
classroom with the Classroom 2000 project at Georgia Tech [1]. They put a Liveboard
in a specially outfitted classroom. Professors would project their notes on it, and they
could annotate them in real time. The notes and annotations were captured, together
with the audio and video of the lecture. The system also supported multiple passive
displays where the lecturer could place material, though they could not annotate it.

While these developments helped the meeting leader/facilitator, other research
efforts worked to provide support for the other people in the meeting. Stewart et
al. defined the Single Display Groupware interface [28]. They designed software for
multiple users with multiple input devices connected to the same display. KidPad,
at the University of Maryland, is the prime example. It uses several novel GUI
widgets to support multiple users interacting with the same drawing surface. Myers
et al. proposed the use of PDAs as the input devices to a shared program. And, the
Classroom 2000 group provided tablet computers, so that students could take their
own private notes along with the lecture.

In all of the recent research setting, pen computing has won out over keyboards,
because keyboards make noise, and pen computing is much more flexible and natural.
Informal observation has noticed that notebooks and on whiteboards contain a lot of
sketching side by side with writing.

Our ideal meeting setup is closest to that of the Classroom 2000, although our
system supports any kind of meeting room. We go beyond its setup by allowing
extensive sharing among the tablet computers and the public displays.
5.2 Evolution of Capture and Retrieval Tools

Capture is perhaps the most important feature of digital meetings, so there have been many schemes to support it. The early systems capture meetings as text documents, so they are easy to browse and search by keyword. Xcapture is a UNIX tool to capture audio in an office [13]. It uses silence detecting algorithms to segment audio, and it shows these segments to the user in a timeline.

We Met from IBM [33], is the first product to consider using handwritten notes as an index to a more detailed meeting record. They created a simple sketching program where users can write anywhere on any number of different pages. Events on the program are logged with undo and redo operations, and so the sequence can be replayed in real time. This interface is very similar to that of the NotePlayer, although it is not linked to the meeting audio.

The program to first combine notes with audio is Filochat from Hewlett Packard [31]. Users take notes in the standard electronic notebook, each note is timestamped, and the notes are combined with the audio recording. Users can play back the audio in the standard way, and they can click on any note to get any point in the audio timeline.

The Coral architecture from Xerox Parc [18] combined their Tivoli note-taking software with an audio/video capture system. The retrieval interface is the same as Filochat, but it includes the more advanced note taking of Tivoli in addition to a video stream. Stifelman’s Audio Notebook [29] works like Filochat, except that it uses real paper on top of a tablet digitizer. The notes and audio are retrieved in the same way, but the note taking is more tangible and natural.

Dynomite is a system from FX Palo Alto to serve as an electronic notebook for individuals taking notes in a variety of settings, not just meetings [32]. It operates similarly to Filochat, except that it does not record continuously, and users can associate properties and keywords with particular notes. This allows not only note browsing, but also retrieval based on these indices.

In the Classroom 2000, lecture recordings are indexed with an automatically created web page. Each slide or whiteboard page that a lecturer makes is stored with
a link to the audio and video that correspond to the time period when the slide was being shown. The lecture slide is also personalized for each user. They see their own notes along with the lecturer’s, and they can slide a timeline to find the audio and video for any particular note. There is also a search feature, so that users can search for words that occurred on particular slides.

Finally, NotePals is a lightweight, note-taking system for PDAs [5]. Users take notes on a standard electronic notebook program. Like Dynomite, users can categorize notes into folders, and they can associate them with keywords. The notes are automatically synced to the user’s personal computer, and they can be shared with other people. So, the public notes of a working group are all shared, and a member can search for them by time, place, person, category, or keyword.

Our method for browsing and retrieving meetings is very similar to the ones above. Users can browse the notes of a particular meeting by a timeline, and they can browse for high level meetings using the Object Browser.

5.3 Evolution of Collaborative Note-Taking Software

Note taking software is a related area with a lot of overlap to meeting capture and support. The work has been about using computation to improve on notebooks: manipulating the notes that are written down and understanding what they imply.

All of the capture tools above use some form of electronic notebook. The standard set of features lets users write anywhere on a page, select sets of sketches, and copy, paste, or delete them. Tivoli [19], the sketching program developed at PARC to run on their Liveboards, is the first program to try to deal with sketches at a higher level. Regular drawing strokes could be interspersed with gestures. Users indicate that a particular stroke is a gesture by pressing a pen button after they draw the stroke. There are special gestures to select sketches, move them around, and scale them. There are also various gestures to create horizontal and vertical space.
Tivoli also contains operations similar to our organizations, associations, and annotations. A user can divide up a page into cells using horizontal and vertical lines, and there is special wrapping functionality to manipulate vertical lists of sketches. Sketches can be minimized and redisplayed, which is somewhat similar to our expanding and contracting of annotations or filters. Tivoli supports association by letting the user draw a link between the circular enclosure of one set of sketches and the circular enclosure of another set.

Tivoli has almost exactly the same goals as the eNotePad: rapid, informal sketching during meetings. The choices for eNotePad’s feature set are validated by Tivoli’s focus on free-form, arbitrary sketches and its support for space creation, page organization, expansion/contraction, sketch grouping, and linking. Tivoli’s interface is more complicated than the eNotePad’s, however, because it does not create any structure for the things that are written on the board. There are no atomic sketches, and there is no hierarchy of containment, annotation, and visibility. The eNotePad’s structure lets users manipulate their notes more simply and intuitively. Also, we prefer context menus to gestures, since the former are arbitrary and hard to remember.

Another generic note taking program is Flatland [21], a system to perform the function that whiteboards perform in individuals’ private offices. These whiteboards typically contain material from several past conversations, and the information has varying levels of permanence and importance. The system segments an electronic whiteboard into an active area and many inactive areas. Segments can be moved around, and each segment can be associated with a particular interpretive domain. For example, there is a domain that performs arithmetic on the board. There is another domain for drawing maps and another for making to-do lists that can be checked off. In contrast to the eNotePad and Tivoli, Flatland is more suited for an individual’s doodling than for collaborative work. It does not create the abstract concept structures that are the products of meetings.

Notebook programs have also moved beyond paper by integrating desktop program functionality onto the notebook surface. The process is generally similar to placing windows on a desktop or visual controls onto a GUI application. Tivoli refers
to these application widgets as domain objects [20]. Domain objects define the set of gestures that can be made on them and the functions that happen when the gestures are made. Their example was a simple spreadsheet application, where the response to one kind of gesture was to add a set of numbers and place the answer at a particular place. The Interactive Mural at Stanford [12] also enhances the standard electronic whiteboard with different kinds of widgets. These widgets respond to special kinds of gestures that are triggered from flow menus, which are a pen-oriented generalization of pie menus. The widgets can even respond to textual parameters, which are recognized from the user’s handwriting. The eNotePad does not have anything similar to domain objects, since we have rarely noticed individuals needing to perform calculations during discussions. However, we do support users copying and pasting images and text into the eNotePad. Also, we support the look-up and annotation of web pages. We believe that this is the extent of the need for desktop objects during meetings for the majority of users.

The final research direction for note taking applications has been the actual interpretation of the notes. Landay and Myers developed the SILK system [16], where user interface designers could roughly sketch out the components of a GUI, and the interpreter attempted to convert the drawing to a simple prototype. Gross and Do developed the Electronic Cocktail Napkin [10], which attempted to understand early designs from a variety of domains such as architecture and electrical engineering. Again, we do not attempt to do any recognition at this level, because meetings generally consist of discussions of very high-level concepts, and the accompanying sketches do not have sufficient detail for the computer to do interesting computation with them beyond simple storage and retrieval.

5.4 Evolution of Meeting Automation

Automation is one problem that the traditional electronic meeting rooms have surprisingly ignored. This is probably because the digital meeting rooms are so expensive to build that the research groups only had a small number of them. So, they had
specialized, ad hoc scripts for each of them, and they did not worry about the problem of rooms describing themselves to personal devices and vice versa. However, we want to enable meeting capture in a large number of spaces with cheap, ordinary hardware. Therefore, for us, infrastructure for capture automation is just as important a problem as the capture/retrieval mechanism itself.

The automation problem can be defined as letting users interact with computers without ever having to connect or configure anything. The host’s agents talk to the user’s agents, and everything that doesn’t require user input should happen without user intervention. This is a problem being tackled in the emerging area of intelligent, context-aware environments (IEs). The goal of these environments is to try to bring computation into humans’ environments, so that their interaction with the computer is completely natural and tangible. A sample of important intelligent environments are EasyLiving from Microsoft Research [25], the AwareHome from Georgia Tech [15], the Intelligent Room at MIT [4], the iRoom at Stanford [14], and the i-Land project at GMD IPSI [30]. Complementary to these special spaces are a variety of custom infrastructures to support them. The major ones are Gaia from UIUC [23], Aura from CMU [26], the Context Toolkit from Georgia Tech [6], Metaglue from MIT [3], and OneWorld from the University of Washington [9].

These environments generally have a perceptual system to generate context and a distributed, application architecture for modeling, using, and reacting to this context. The context perceivers generally consist of a mix of multimodal person trackers and identifiers, activity recognizers, and spoken dialogue systems. There has been a large amount of work in this area. However, because we are only concerned with the proper use and distribution of context, we do not care how it is generated. Since good perceptual systems are challenging to develop, we generally use ordinary GUI dialogues to get the information we need.

For context modeling and reaction, most of the systems act in a similar way. There is some sort of dynamic representation of the capabilities of a space using either tuples or different XML schema.

The infrastructures for supporting individual intelligent rooms generally extend
the concepts of operating systems for desktop computers to rooms. There is a standard set of services such as a catalog of hardware, a permanent, file store, and a publish-subscribe event queue. Gaia and the iRoom use tuples to model device and user context, as we do. The problem with all of the environmental infrastructures is that they only bring the "single computer" abstraction to a room. From the user's perspective, the entire world should appear to be one computer, and their information, preferences, and activities should follow them everywhere. This motivates our universal object catalog and our distributed object-oriented database network.
Chapter 6

Conclusions and Future Work

In this thesis, we have described our collaboration and meeting support system. We believe that such a system is easy to use and will significantly improve a group's synergy. From our past failures, we believe we understand what a meeting support system must provide, and we believe we now have the infrastructure to provide it.

The next step of course is to finish the implementation, because this is the only way to test our theories of interaction. There is a phenomenon that Jonathan Grudin has described and that A'bdowd has discovered in Classroom 2000 [11]. Users are less accepting of new forms of groupware if it means that they have to change the way they are comfortable doing things. So, students in Classroom 2000 loved the interactive whiteboards and the capture record that they could browse from their web browsers. They were less enthusiastic about the electronic tablets, which required effort to learn to use and which detracted from the paper notebooks they were used to.

We foresee that we will have the same problems as we try to make more aspects of collaboration electronic. However, we look forward to the challenges that these problems present and the hints they will provide to help us design more natural and invisible user interfaces.

There are many things that we want to do once we have a user population regularly recording their meetings. First, we want to generate a textual record of the meeting using the materials presented and through handwriting recognition of the written notes. From this textual record, we can build a search engine that can scan an
archive of meetings for particular keywords.

Handwriting recognition can be successful at classifying a meeting even with significant recognition errors. The errors a handwriting recognizer makes are gibberish words that no one searches for anyway. In contrast, when a speech recognizer makes errors, it creates a transcript of real, false words, which could seriously impede the precision of an information retrieval system. Nevertheless, speech recognition is still very important. If we had perfect speech recognition, the spoken transcript would be the most important meeting artifact we would generate. To generate a transcript with a reasonably small number of errors, we will investigate putting various close microphones onto the meeting leader, and we will attempt to recognize only their speech with a speaker dependent, large vocabulary model.

While this work has focused on the capture of the note stream, it is also important to automatically capture a high quality stream from the audio and video. For audio, capture involves finding the closest microphone to whoever is speaking, possibly even beamforming from multiple microphones to further eliminate background noise. Capturing video requires finding the most important view out of multiple cameras and switching to it. There are two popular approaches that can be combined in some ratio. One method is to use a face detector to determine what the majority of people in the meeting are looking at. Another method is to find the location in the room producing the most noise.

At the higher, symbolic level, we have not exhausted the possibilities of the Object Browser. This program is part of a growing set of programs for mining and interacting with the Semantic Net. The Semantic Net is the consensus engine for the next generation of Internet technology. However, there is not yet a rich, simple interface like a web browser for navigating and searching this structured, interactive network. Such an interface would be quite valuable in helping to spread the growth of the semantic net. It would empower web developers to more easily write and deploy interactive programs that take advantage of the user's context. And, it would empower users to personalize and automate their interaction with the net with agents. That is, if the content on the net is self-describing, standardized, and interactive, agents can
be much more helpful at bringing useful functionality to their user. To improve our Object Browser system, we want to explore ways we can build a search engine for the semantic net, and we want to explore ways for users to train and program their own agents. Chart 1 shows a sample of settings and use scenarios that we would like to explore.

In addition to trying to mine our capture data, we will also attempt to move our software beyond the meeting room. Specifically, we would like to use it in classrooms and remote meetings. We would also like to try to use it to support small, informal meetings that don't occur in official meeting spaces. These spontaneous interactions generally take place in offices or transitional spaces. It is difficult to equip these areas with computation, because they occur in such unpredictable places, and they are so unstructured. In such settings, electronic whiteboards and notebooks are a nuisance compared to their passive ancestors. The best we can hope for is to try to capture the raw activity. The recognition and retrieval problem for these records will be much more difficult problem than the one for formal meetings.

Finally, we must confront the issues of security and privacy. When a user's personal information and activities can be accessed from anywhere, we must provide safeguards, so that we can guarantee that the user authorized this access. This also applies to privacy. Records of meetings and conversations should only be captured at users' consent. Before meeting capture can be implemented widely, we must create strong mechanisms so users have complete control over their activities that are captured. Ubiquitous computing promises to interconnect a much broader swath of human life than computers have touched before. This has the potential to generate many exciting new technologies, but it also burdens us with the heavy responsibility to make sure that this computational power can only be used for beneficial purposes.
Bibliography


[28] Jason Stewart, Benjamin B. Bederson, and Allison Druin. Single display groupware: a model for co-present collaboration. In *Proceedings of the SIGCHI con-


