BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-92-M13

"PIEmail:
A Personalized Information Environment Mail Tool"

by

Bob Weiner

PIEmail:
A Personalized Information Environment Mail Tool

Bob Weiner
Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the
Degree of Master of Science in the Department of Computer Science
at Brown University

May 10, 1992

The author's work on this project has been sponsored by Motorola Inc. under the Distinguished Student-Employee program.

This research project by Bob Weiner is accepted in its present form
by the Department of Computer Science at Brown University
in parital fulfillment of the requirements for the Degree of Master of Science.

Peter Wegner

Professor Peter Wegner
Advisor

May 10 1992

Date

Abstract

PIEmail is one of a new generation of electronic mail (e-mail) handling tools that integrate mail into the day-to-day information handling and communication environments of modern professionals. In addition to standard sequential message processing features, PIEmail offers automated message classification into folders, queries over message bases, task and role-based mail addresses, button-based hypertext facilities, and an extensible mail processing library.

We discuss the requirements for and design of PIEmail and illustrate its operation through a number of example uses. The design is specified through a series of object-oriented class-relationship diagrams together with discussion of major classes.

PIEmail hyperlinks are provided through integration with the Hyperbole associative information manager [*Weiner92*]. We discuss the Hyperbole model and how it works together with PIEmail.

PIEmail is an initial step towards a larger Personalized Information Environment (PIE) model. We briefly outline our view of a PIE architecture.

Finally, we examine major prior related work.

Table of Contents

1 Introduction

Electronic mail has emerged as one of the communications media of choice among technical professionals. Most present day e-mail tools have been designed with low volumes of mail in mind, under the assumption that e-mail represents just a small fraction of the information exchange that occurs. The torrents of e-mail that people are likely to receive in the coming decades require a fundamental change in e-mail reader design to permit effective message management and integration within an electronic workspace [*Mackay88*]. PIEmail is a design that confronts these issues.

PIEmail reads standard UNIX mailboxes and files them according to personal rule sets into folders. Folders may be browsed and handled in a similar fashion to the way net newsgroups are today [Horton83]. Rules may also annotate the messages with desired attributes such as priorities and expiration dates. Mail is then read by selecting folders from a list and viewing the messages inside. A message summary view may also be used to rapidly scan and process the messages in each folder.

To further simplify the viewing of folder contents, messages may be sorted on the fields in their headers or any attributes they have. They may also be filtered to provide a view of a subset of the messages. A conversation view may be used to follow the flow of dialogue across messages in a folder. (We may at a later time, add tickler reminders that prompt you to follow up on conversations by mailing a specified message from the conversation to you on a given date.) Queries may be made over a set of unopened folders to retrieve messages that match particular criteria.

A rolodex helps manage mail addresses so that when composing mail, addresses are always at hand. PIEmail's integration with the Hyperbole associative information manager supports easy inclusion of hyperlinks in messages through the use of hyper-buttons that may be mailed across local and wide area networks, see Chapter 5 [Hyperbole], page 29. PIEmail also utilizes Hyperbole's Smart Key context-sensitive mouse handling support to provide direct selection of entities within PIEmail views.

These facilities help to significantly reduce the problem of mail overload that many people experience today when mail from sources all around the world, on a myriad of topics, possibly over a number of days, flows into the one and only mail box that they have been assigned. While their computers sit idle, the mail waits in this box. When they return after a few hour or few day hiatus, they are confronted with an overwhelming calvalcade of unrelated messages which they must sift through and respond to sequentially. Large message sets often slow mail readers greatly, making this an even more frustrating task.

The rest of this report is organized as follows. Section 2.1 [PIEmail Operation], page 7, includes a more detailed functional summary of PIEmail. Section 3.4.2 [PIEmail Requirements], page 18, then covers system requirements before Chapter 4 [PIEmail Design], page 19, delves into the detailed design of PIEmail. Chapter 5 [Hyperbole], page 29, discusses Hyperbole, our associative information manager integrated with PIEmail. Our broader view of PIEs are presented in Section 6.3.3 [PIE Model], page 52. Finally, we compare our work with prior published work in Chapter 7 [Prior Work], page 57.

The diagrammatic notation used throughout the report is explained in, Appendix B [Class Diagrams], page 73. See Appendix A [Glossary], page 71, for a glossary of PIEmail related terms. The chapter on Hyperbole has its own glossary, however. See Section 5.6 [Hyperbole Glossary], page 45. If any GNU Emacs terms are unfamiliar to you, see section "Glossary" in *the GNU Emacs Manual*. A basic familiarity with the Emacs editing model as explained in [*Stallman87*] is useful when reading this document. The material covered in the GNU Emacs tutorial, normally bound to {C-h t} within Emacs, is more than sufficient as background.

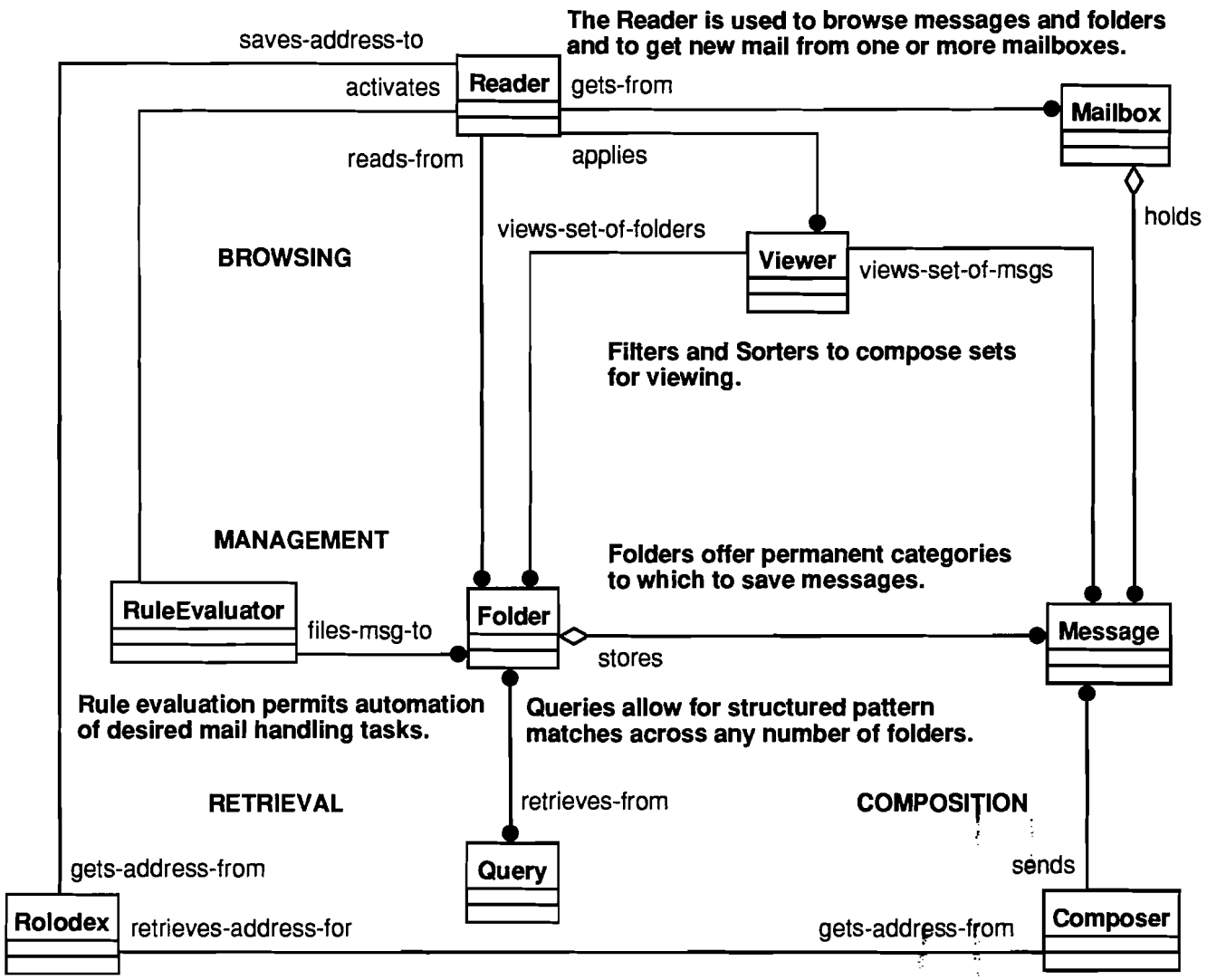
This is a technical report on PIEmail, not a user manual. As such, it provides only a cursory view of how one would actually operate the PIEmail reader. Another document will eventually fill this other need, leading the user step by step through all of PIEmail's operational specifics.

2 PIEmail Operation

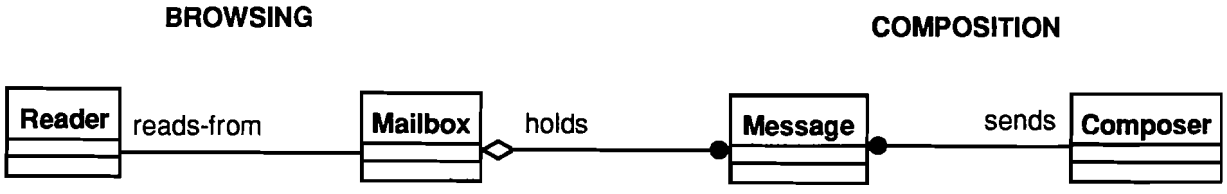
This chapter gives basic details on PIEmail operation. Complete details are to be included in a later user manual. Organization, invocation, message handling and retrieval, and address lookup are presented.

First, let us look at the conceptual organization of PIEmail and contrast it with a traditional mail reader, like the UNIX `mail` command.

PIEmail Concept Bob Weiner 5/10/92



Traditional Mail Reader Concept



The main functional areas of PIEmail are broken out in the diagram: browsing, management, retrieval, and composition. Traditional systems offer fixed message browsing and message subject summary views. PIEmail allows user configuration of message browsing and summary views thereby permitting more flexible perspectives on message bases.

Traditional systems offer little mail management support, so users tend to accumulate large in-boxes, often with hundreds of unorganized messages. PIEmail uses the folder as the central message organization construct. Priorities and other message attributes also help in message organization. A folder list view further supports folder-level management.

Traditional systems also lack message retrieval support. PIEmail provides form-based queries for searching over large message bases.

Traditonal systems offer a single mail composition form with no means of providing targeted forms to meet various messaging needs. PIEmail supports selection from among user or sited defined composition forms. It also has an integrated rolodex which can rapidly retrieve addresses for use during mail composition, so one is not burdened with remembering addresses or having to create aliased names for each user.

2.1 Interaction Techniques

PIEmail offers three overlapping means of reading mail, so that users may use any and all means they find comfortable.

1. Direct manipulation [*Shneiderman82*] may be used. Folders from the folder list may be selected to open them and to summarize the messages therein. Selection of a message summary item displays the associated message. A click on a Hyperbole button activates it. One can also click at the end or beginning of a message to move to the next or previous message.
2. Commands may be invoked directly from the keyboard as in traditional mail readers.
3. Commands may also be selected from menus, minimizing the need to learn any command set.

2.2 Mail Reading

PIEmail is built atop the GNU Emacs editor. So when using PIEmail interactively, one first starts GNU Emacs and then invokes PIEmail to read mail. PIEmail can be started by selecting a menu entry or with **{M-x pm RTN}**. With default settings, it reads any waiting mail and displays a list of available folders. To read individual messages, one simply selects a folder from the folder list. (Initially, there is only one folder. A user or his automated mail filing rules may create others.) Selection of a folder opens the folder and displays the first unseen message or the last seen message, if all messages have been viewed before. All folder and message commands may be applied from this view. One can move to a previous or next message or jump to a message by number. One can also delete, undelete, or edit messages. A message may be copied from one folder to another or printed. Each folder can have any number of mailboxes attached, which will be read whenever the folder is opened.

```
                                Reader Menu
-------------------------------------------------------------------------
Help                        Print-Msg               Next-Screen
Prev-Screen                 Next-Msg                Prev-Msg
Next-All-Msgs               Prev-All-Msgs           Summary-Of-Msg-Hdrs
First-Msg                   Last-Msg                Jump-To-Msg-Num
To-Msg-End                  To-Msg-Begin            Del-Msg-To-Next
Del-Msg-To-Prev             Undel-Prev-Msg          Expunge-Deleted-Msgs
Expunge-And-Save-Msgs       Edit-Msg                Get-New-Mail
Reply-To-Sender/            Reply-To-Everybody/     Forward-Msg-To-Addr/
Send-Mail/                  Continue-Send-Mail/     Append-Msg-To-Unix-Mail
Append-Msg-To-Folder        Read-Other-Folder       Add-Msg-Attribute
Kill-Msg-Attribute          Filter-By-Attribute     Toggle-Show-Short-Msg-Hdr
Quit-Reading-Mail           <Quit-Menus>            <Top-Menu>/
<Last-Menu>
-------------------------------------------------------------------------
```

Items that end with '/' invoke submenus. Items may be selected with the mouse or from the keyboard via the Smart Key. See Section 5.5.1 [Smart Keys], page 38. Each menu item may offer its own help.

The first menu item, `Help` displays a section of the PIEmail documentation that explains menu operation. Succeeding items support moving throughout a folder. These are followed by deletion commands and then message composure commands. A few folder operations are included, followed by some attribute commands and finally some menu control items.

Menus are, in fact, available to control many aspects of PIEmail operation, with keyboard alternatives also offered. The construct menu described later gives direct access to submenus that manage PIEmail facilities. Hyperbole has its own menus, see Section 5.5.2 [Operating Menus], page 41. Menus of Hyperbole buttons may also be constructed to perform personalized actions.

2.3 Mail Summaries

A summary of the folder's message headers may be generated with the `(pm-msgsumm-create)` command bound to {h}, for header summary. Most of the message commands from the folder buffer may be given from the summary buffer. The folder and summary views are often on screen at the same time, so the effect of an operation in one is immediately reflected in the other.

The summary format is wholly user configurable, except that it must include message numbers at the start. It typically contains the sender's address, the date sent, and the subject. Here is a personal sample:

```
 Num  Date       Priority  Lines   Sender              Subject
-------------------------------------------------------------------------
   1  25-Mar-92  High     143 Ln  kurt@Think.COM      Electronic books
   2  30-Mar-92           10 Ln  To: rsw.pie         Here's a <(button)>
-> 3  01-Apr-92  Urgent     6 Ln  To: rsw             Meeting at 1pm
   4  01-Apr-92  Trivial   34 Ln  hws@csis.dit.csiro. Assertions in Sather
   5  01-Apr-92  Pr: 6      6 Ln  To: rsw             Priority example
```

This table illustrates many PIEmail facilities. The pointer on the extreme left indicates that the currently selected message is number 3. Three fields over we have the Priority field. It may be set by message categorization rules or interactively, after a message is read. Priorities may be given as names (messages 1, 3, and 4) or as numbers (message 5). Each priority name equates to a

number so that messages may be ordered by priority. A priority value of 6 has no name equivalent so it is displayed numerically with the "Pr:" prefix (message 5).

The Lines field indicates the message length to help one determine whether to read lengthy messages or to save them for later. The number of bytes in the message may also be shown.

The Sender field shows the orginator of the message (messages 1 and 4) except when the message sender id is the same as the reciever's user id (messages 2, 3, and 5). In those cases, the recipient field is shown instead with a "To:" prefix. This is most useful when the addressee is different than the user id. Message 2, for example, was sent to a task address, rsw.pie, used to deal specifically with PIE issues. Another useful case is when the mail is to a mail list of which the sender is a member. The To address might be "To: PIEmail-users". By placing that mail list name in the summary with the "To:" prefix, it stands out a bit. Then one knows that it is one of your own messages and can be ignored or only scanned briefly.

The final field is the Subject field. Message 2 includes a Hyperbole button in its subject field. The button may be activated from either the message summary or the message view. It could provide a link to a remotely stored file or directory. It could open a folder or demonstrate other PIEmail facilities. See Chapter 5 [Hyperbole], page 29.

2.4 Message Ordering and Filtering

Messages are stored in folders as they are read in from a mailbox. Folders can grow larger than one may want for particular tasks. Often one wants to see them in a different order or look at just a subset of available messages. PIEmail offers sorters and filters to provide alternate views of the messages within a folder. Built-in sorters include: by date (time of mailing), by sender, by receiver, by subject or conversation, and by priority. Other sorters may be defined with a small amount of programming and easily installed into PIEmail.
The sorters menu enables management and application of sorters.

```
Sorters Menu              Descriptions
-------------------------------------------------------------------------
Apply                   - Applies a sorter to current view
Create                  - Creates a new message sorter
Delete                  - Deletes a message sorter
Edit                    - Edits a sorter
Rename                  - Renames a sorter
```

Filters apply a predicate over a set of messages and return either the ones matching the predicate or the ones that failed to match (by inverting the predicate). Built-in filters include: subject, attribute (any message attribute, such as priority), addressee (sender or receiver), all (to restore all the messages from the folder) and date range filtering. Other filters are easy to write. A personalized example might be to filter out all messages in a folder except those from local co-workers. This would provide the local perspective on the topic so that it could be compared to external inputs.

The Filters menu offers the these commands:

```
Filters Menu              Descriptions
-------------------------------------------------------------------------
Apply                   - Applies a filter to current view
Create                  - Creates a new filter
Delete                  - Deletes a filter
Edit                    - Edits a filter specification
Rename                  - Names or renames a filter
```

Filtering may also be done manually by individually marking messages from a message summary buffer and then hiding either the marked or unmarked messages.

Filtering and sorting a folder produces an ordered subset of the messages in the folder. PIEmail therefore displays two message numbers for filtered or re-ordered messages. One is the absolute number of the message indicating its physical order and another is a relative number giving its position in the ordered set.

Many common PIEmail message commands may be applied to message sets produced by filters and sorters or to entire folders. Deletion, copying, archiving, printing, and setting or clearing an attribute such as priority, may all be done to message sets. If a new summary is produced from the folder, it reflects only the messages in the current set. Thus, sets provide an efficient way of quickly processing large numbers of messages, moving away from the traditional sequential processing of traditional e-mail systems.

2.5 Folder Use

Messages may be grouped into folders, either manually or automatically according to rule sets, see Section 4.7.2 [Rules], page 26. All messages pertaining to a particular conversation are usually put into the same folder. Folders thereby provide a natural reference point from which to seek out topic-based information.

PIEmail supports the nesting of subfolders and allows for the storage of arbitrary, user-defined attributes in addition to a standard set of per-folder attributes.
The Groupings menu handles groupings of messages stored as folders. It offers these commands.

```
Groupings Menu            Descriptions
-------------------------------------------------------------------------
Attribute               - Sets or clears a folder-specific attribute
Create                  - Creates a folder (message grouping)
Delete                  - Removes a folder and all of its contents
List                    - Displays available folders
Open                    - Reads messages in a folder
Rename                  - Names or renames a folder
View/                   - Folder list view commands
```

2.6 Message Composition

Outgoing mail messages are composed for three reasons.

1. One wants to compose a new message for mailing.
2. A reply needs to be sent in response to a message.
3. A message should be forwarded to other parties and annotated before it is sent.

Message forms are used as templates during mail composition to support these differing needs. Forms allow one to have templates for different kinds of messages: one for meeting notices, one for

mailing out reports, one for forwarded messages, etc. Each form can have any number of blank and already filled-in fields but its header portion must conform to any message format standards that the message transfer agents require. For example, under RFC-822, the basic Internet message format standard, non-standard field names should be preceded with an "X-", as in "X-My-Field-Name: my-field-value", to ensure they do not conflict with standard names [*Crocker82*].

Pm-msgform-forward specifies the form to use when forwarding a message. *Pm-msgform-reply* gives the reply form. *Pm-msgform-mail* specifies the default form to use whenever the mail composure command is invoked with `{C-x m}`. With a prefix argument, `{C-u C-x m}`, one is prompted for a form to use.

The list of available forms is composed from a list of forms available at the site and from a personal msgform directory, if the user has one. Personal forms are used in preference to site forms whenever there is a form with the same name in the personal directory and the site directory.

2.7 Query Operation

Filters apply pre-determined predicates to opened folders. Queries on the other hand are generated interactively and are used across any number of opened or unopened folders. Queries are generated by filling out a form much like entering an outgoing mail message. Instead of sending mail, however, completed query forms are used as search criteria to locate matching messages across a set of folders. One fills in the particular information known, such as key words, a sender address, and the message sets over which to search. Blank fields match to anything, including non-existence of the field. Here is an example.

```
---------- Query Form ----------
Search: +thesis, +stock-market
Sort by:
From:
To:
Subject: mail && information
Dates:   1-1-92 -
Attributes: seen, !deleted
Body:
--------------------------------------------------------
```

We want to to look in the thesis and stock-market folders for all messages whose subject lines contain both of the terms, mail and information, in either order. The '+' character at the start of the items in the Search: field indicates that they are folders to be found in our folder directory. Other paths may be searched by prefacing them with a '/' or '~' character.

We further restrict matches to messages from this year which have already been seen and are not marked for deletion. Here is the query result.

```
--------------------  Query Results  ---------------------
============================================================
Folder-or-path     Msg-num   Sender      Date      Priority
   Subject
============================================================
+thesis              7       sfh         1-05-92   High
   Update on PIEmail and information environments work.
+thesis             55       pw          1-05-92   Urgent
   Review of your PIEmail thesis information.
+stock-market       39       mkt-anal    2-02-92   Low
   Apple provides agent-based electronic mail information.
+stock-market       46       mkt-anal    2-05-92   Low
   Information Lens intelligent mail processor.
------------------------------------------------------------
```

The messages that match the query are called hits. Hits are identified in summary form by location and message number. Any hit summary line may then be selected for display. One may also walk through the hits sequentially, essentially treating the items as a folder summary. The results may also be sorted by date or priority and so forth, either by filling in the "Sort By:" field as part of the query or interactively, in the same fashion as for folder summaries.

2.8 Rolodex Use

A full, menu-based rolodex that manages e-mail addresses and contact information is integrated with PIEmail. It supports the storage of hierarchical records such as:

```
* Company                                              1-800-555-5555
** Manager, John       <manager@company.com>           1-212-555-8920
***  Underling, Roy    <underling@company.com>         1-212-555-8923
        Any sort of contact information can be placed here since
        records are free form and may be any number of lines.
```

Retrieval of any record retrieves its entire subtree. Thus, searching for Manager turns up all of his underlings. Searching for Company retrieves all listed employees.

Names are stored in last name, first name order for ease of sorting. Records are otherwise free form, except that e-mail addresses are normally delimited with < and > symbols for ease of location. (They may, however, be recognized heuristically by their format if the delimiters are not used.) Hyperbole buttons may also be included in records.

Records may be added, edited, searched for by string or regular expression (see section "Regular Expressions" in *the GNU Emacs Manual*) and inserted into the current editing buffer.
The rolodex Addresses menu provides access to the following commands:

```
Addresses Menu          Descriptions
-------------------------------------------------------------------------
Add                   - Adds an entry to the rolodex
Display-again         - Redisplays matches from last search
Edit                  - Edits an existing rolodex entry
Kill                  - Removes an entry from the rolodex
Order                 - Sorts all levels in rolodex
Regexp-find           - Finds all matches for a regular expression
String-find           - Finds all matches for a string
Yank                  - Inserts first matching rolodex entry
```

For any of these commands that prompt you for a name, you may use the form parent/child

to locate a child entry below a parent entry, e.g. from the example near the top, we could give Company/Manager/Underling.

If an add request is made while in a folder message buffer, the sender's name and e-mail address are automatically selected as the default record to add. If the default is accepted, the name together with its e-mail address will be added, after which the user may freely annotate the entry.

While composing mail, one may enter a name or name fragment and then request that it be replaced by any associated e-mail address. This saves one from having to make extensive alias lists to make addresses easier to remember. Personal e-mail aliases from '`.mailrc`' files and active local sendmail aliases may also be expanded to check that the proper addresses are being used.

Rolodex searches are simple and rapid. One selects string or regular expression search from the menu and enters a search key. Any matches found are displayed in another window. If they contain Hyperbole buttons, the buttons may be activated directly from the match buffer. For example, one could attach a link button to a sofware author's entry which pointed to his software archive. Then one could retrieve the link by searching for the author, the software package name included in text of the entry, or the button label.

2.9 Construct Management

The Constructs menu is available to manage high-level PIEmail constructs. It contains the following items:

```
Construct Menu            Descriptions
-------------------------------------------------------------------------
Addresses/ (Rolodex)    - Address and contact information manager
Buttons/                - Hypertextual associative link buttons
Conversations/          - Sets of messages that reference one another
Documents/              - PIEmail-related online documents
Filters/                - Grouping filters that alter the set of messages
Groupings/ (Folders)    - Persistent groupings of messages
Messages/               - E-mail messages with header and body structure
Queries/                - Interactive retrievals across folders
Rules/                  - PIEmail automation condition-action support
Sorters/                - Grouping ordering according to criteria
Tasks/                  - Task or role-based personalized addresses
Views/                  - Specifications for PIEmail entity display
```

2.10 Document Reference

The documents menu can be used to browse PIEmail documentation when one is looking for specific operational or design details.

```
Documents Menu            Descriptions
-------------------------------------------------------------------------
Copyright               - Displays PIEmail Emacs-type copyright information
Glossary                - Displays PIEmail glossary of terms
InfoManual              - Displays PIEmail user manual in Info viewer
Manifest                - Lists files in PIEmail distribution
Report                  - This report covering PIEmail design
```

3 PIEmail Requirements

Now that we have seen what PIEmail can do, we back up a bit to examine the requirements behind it design. As indicated earlier, we see a mail reader's basic mission as a means of browsing, management, composition and retrieval of messages. Browsing requires a means of viewing messages, generally from a number of different perspectives gained from sorting messages into different categories and orderings. One also has to be able to find saved messages according to known criteria.

Message management should permit logging of outgoing messages, tagging of messages with attributes, as well as message deletion and refiling. The ability to work with groups of messages at once is also desirable.

Composition requires editing facilities and the ability to include external information for mailing.

Retrieval necessitates searching over groups of messages. It can also mean access to an address book to look up mailing addresses from names or vice versa,

A mail reader capable of personalization has further requirements. Commands should be automatable and the system programmable to allow for personalized tailoring. One should be able to filter messages and to produce custom views over a message base. A number of personal mail addresses should be allowed to support a diversity of work tasks. The reader's user interface should be multi-modal, essentially offering multiple interfaces, so that it can work for people with different skill sets and adapt as those skills change over time.

In the following sections, we describe in further detail the requirements behind the PIEmail concept.

3.1 Browsing

Electronic mail tools are often called mail readers, indicating that their primary purpose is to display mail for viewing. But the views that mail readers provide their users are often quite rigid, essentially coded into the reader itself. PIEmail seeks to give users better control and usage of their mail by letting them customize standard views and build their own.

3.1.1 Form Usage

Form support should permit the specification of field-based forms together with the behavior needed to fill them out. At a minimum, one should be able to constrain the header fields visible when messages are displayed and to alter the field layout in message summary views. One should be able to create forms for particular message composure tasks, e.g. a meeting notice form.

3.1.2 Viewing

Facilities for browsing of available message categories and the messages filed within each category should be offered along with summaries of messages within a category. Standard mail handling functions like deletion need to be included.

One should be able to move quickly from one category to another and to have multiple categories open for browsing at the same time.

3.1.3 Structure Handling

PIEmail should treat messages as semistructured entities [*MaGrLa87*]. That is, it should make use of particular structural requirements that messages must satisfy and allow other parts to be free-form. Messages are composed from a series of header fields together with their values. Message headers are followed by a possibly empty body. The body may contain multiple parts if the header contains a *multipart* valued `Content-type` header field.

3.1.4 Address Capture

When a message comes in from someone unfamiliar, one should be able to quickly add his name and mailing address information to a personal address book. The address book should also permit free annotation of this information with additional material.

3.2 Management

Much of the value we place on information comes from its organization. Organization influences information access, quality, and utility. In the messaging domain, senders supply the first level of organization, the message. Receivers can exercise control over all higher levels, and can alter received messages to their liking, if need be. This section discusses PIEmail requirements for managing groups of messages.

3.2.1 Categorizations

One should be able to categorize messages, to prioritize them, to reorder them, to produce groups of messages that match particular conditions and to tag messages with standard or personalized labels.

Categorization is critical to effective message management and location. As message volume increases, it is an all too common occurrence for important messages to become buried. Prioritization keeps unimportant messages from obscuring high priority ones that are categorized together. By sorting on priority, one can quickly see the messages that must be handled first. Further, one should be able to filter out messages that are irrelevant for the moment. One can then get a clear view of particular messages, e.g. those from your boss.

Another technique useful for filtering and grouping is to label messages with particular tags. Priorities are a special case of this technique. One can then search on and group by the label name whenever necessary.

3.2.2 Conversations

Conversation summaries should be offered so one can see how discussion has developed, i.e. who replied to whom, and what was said. Lack of conversation tracking has long plagued electronic mail tools. It is a chicken and egg problem. Due to the weakness of e-mail readers, people often try to process and delete messages quickly to keep from being inundated. Then it becomes impossible to track the conversation because the messages are gone. A richer set of views of message-bases reduces the need to remove messages permanently, permitting more comprehensive tracking of inter-message relations.

3.2.3 Sets

Most operations available for individual messages should be available for sets of messages. This seems a natural requirement since our aim is to reduce the amount of time spent processing mail while increasing the utility of messaging. When scanning message summaries, one often finds a number of messages that need to be processed in the same way. If one could group the messages and then apply a procedure to the group, this could frequently save a good deal of time, not to mention the immense mental distraction of repeating an operation over and over again.

3.2.4 Task Support

Users should be able to create their own mail addresses in a way that supports project-oriented work but does not permit spoofing of another user. We want to put mail handling control into the hands of the receiver rather than the sender. A user could give a personally generated address to a working group he needs to correspond with to help ensure that all mail from the group, regardless of subject, is processed the same way.

On the other hand, we don't want users to be able to impersonate one another or to acquire privledges that they should not have. So we must limit the form of a personalized address to something that still identifies the party whose address it is.

3.2.5 Automation

User-level commands should be automatable. A common failing of the easier to use graphical environments of today is that there is seldom a way to automate the functions that they offer. For testing and oft repeated operations, one then has no choice but to interact with the programs and their required tedium. PIEmail should offer a support structure for automating commands and a programming library for extending its functionality.

PIEmail must offer a means of pre-processing messages before they are read, to allow pre-categorization, or even deletion or forwarding.

3.3 Composition

3.3.1 Editing

Editing is fundamental to effective information creation. PIEmail is built within the GNU Emacs editor because of its strong base of editing operations and its platform portability. Emacs editing operations such as how to search through texts and how to include a file's contents into a buffer should be available and natural to use within PIEmail.

3.3.2 Hyperlinking

Hyperlinks should be available so that information may at times be referenced within messages rather than included, to minimize message size and the amount of out-of-date information sent.

If a message is sent to a mail list with a number of attachments, there is a good likelihood that a number of the list members will just throw it away, mostly unread. If links rather than attachments

were sent, only those interested would follow the links and use the bandwidth necessary to transfer the information to their local display. In the long run, such a strategy could represent large savings in communications. Since many mail transfer agents limit the size of messages, this would offer a solution that permits the mailing of messages linked into wide-area information bases without the need to transfer all of the associated information in the message.

Both textual header fields and textual body components should be able to carry hypertext buttons that link to external information or that perform other actions.

3.4 Retrieval

Retrieval facilities can help motivate efforts at message organization. Organizational work is worthwhile if one can then locate desired information when needed. Full-text searching is the preferred retrieval technique in messaging domains because the known information used for searches may often be scanty. For those times where one knows which fields particular information occurs in, structured searches should also be permitted.

3.4.1 Querying

Queries should be able to search large message bases and to display summaries of matching messages. One should be able to traverse these summaries looking for messages of interest. Queries should normally be given by filling out a query form with known information to match against. The form should be very similar to the basic mail composition form. Queries should return a summary listing of matching messages which can be browsed to locate desired messages.

There should be a means of saving queries for application at a later time. For example, one may want to produce archives of messages once a quarter where only the prior quarter's messages are saved. A query could be stored to do this where one would simply fill in the quarter desired. The forms used for queries should therefore be customizable.

3.4.2 Address Lookup

Mail addresses and aliases should be easy to lookup and to insert into outgoing messages. Given a name, one should be able to ask for the person's mail address only or his entire record of information from an address book.

4 PIEmail Design

This chapter presents a class-structured view of PIEmail, examining in turn its main constructs and personalization facilities. PIEmail's approach is to provide a strong, extensible information management base tailored to the messaging domain, which may be highly personalized to improve its utility to individual and organizational users.

Two interfaces are being developed for PIEmail concurrently, based on the same general PIEmail design. One runs under GNU Emacs [*Stallman87*] and may be used on terminals and UNIX workstations. (Under the X window version of Emacs called Epoch [*KaLoCaLa92*], it offers a number of enhanced display features.) The other implementation offers a more refined user interface under the NeXTstep environment on a NeXT computer [*Hamlyn92*]. This document is concerned solely with the Emacs-based version.

4.1 PIEmail Invocation

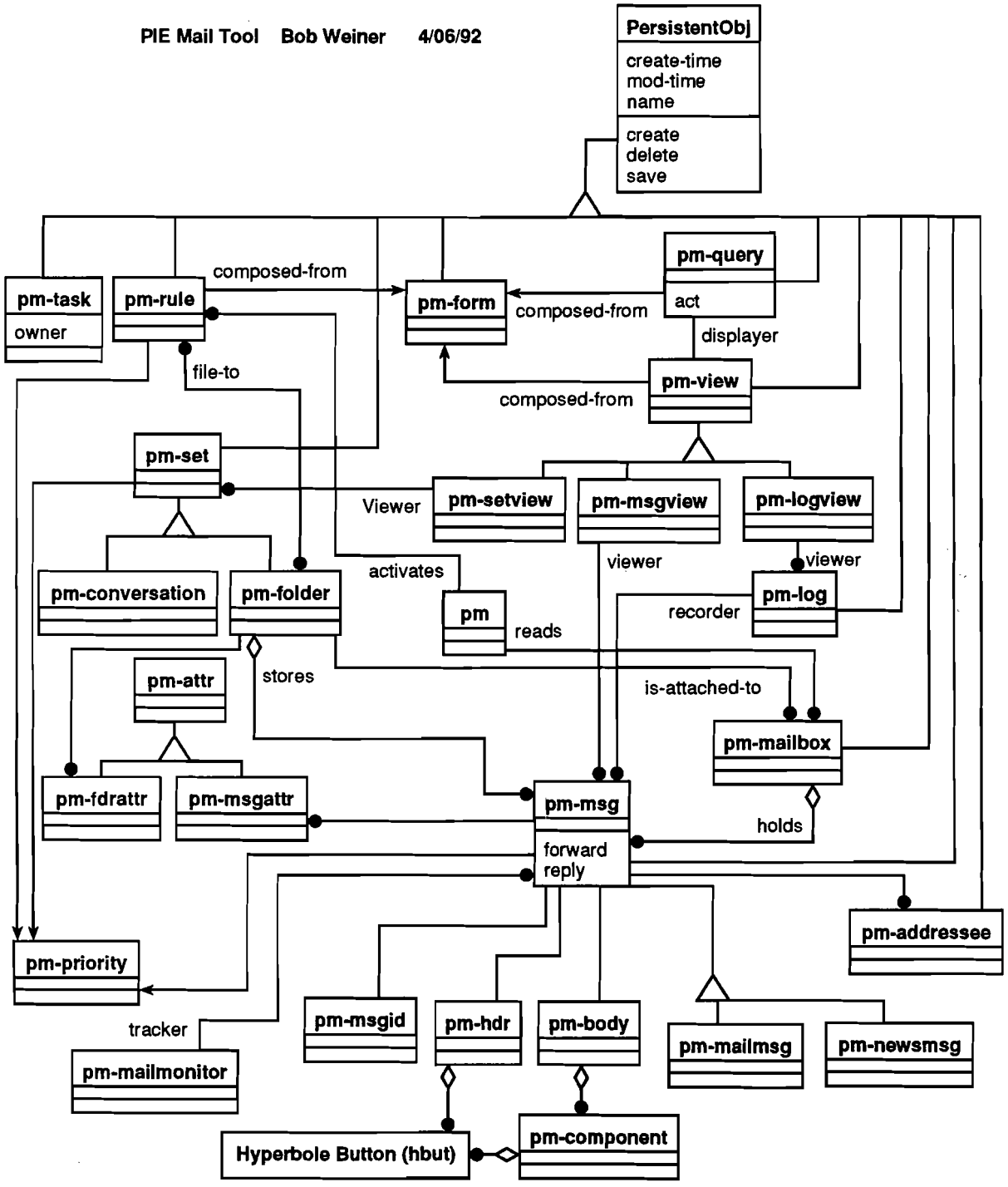
The `pm` class provides interactive entry, exit and configuration points to PIEmail. It maintains state such as the directories to which things are filed, the PIEmail version number and many other configuration settings. The `pm-version-info` command displays an extensive listing of PIEmail settings. `Pm` also contains methods which invoke PIEmail, read new mail, file into folders according to personalized rule sets and display via chosen views, and quit from the mail reader.

Message editing is a straightforward process since PIEmail operates within the GNU Emacs editor. One simply requests that a message be edited; it is then displayed in an editable fashion. A special `pm-msg-edit-mode` is used which provides all regular editing facilities plus two additional commands: one which translates the results of the edit back to the message and another which aborts the edit.

4.2 Entities

PIE entities are the informational artifacts managed by a PIE. All entities share certain base behavior. Entities may be named and thenceforth referred to by name in addition to any standard means by which they can be addressed. Entities share a base protocol of common operation names:

`add`	add an entity to a set
`copy`	duplicate an entity
`create`	create a new instance of an entity
`delete`	mark for deletion or actually delete
`link`	link to this entity from some source
`move`	change the location of an entity
`remove`	remove an entity from a set
`rename`	change an entity's name (give it one if it has none)



4.3 Mailboxes

The `pm-mailbox` class provides access to mailboxes and their contents. PIEmail may eventu-

ally support access to mailboxes on remote machines (relative to the one where it executes), so encapsulation of mailbox access with a class can hide from a PIEmail client the details of whether a mailbox is local or remote.

`Pm-mailbox` provides methods to determine the size of a mailbox, its message count, its format, and to retrieve all or a portion of its messages.

4.4 Folders

Mailboxes are locations into which mail transfer agents deposit mail messages. PIEmail, like all PIE tools, abstracts away from the underlying data formats that it must deal with to provide uniform access to messages in a variety of formats. Specifically, PIEmail can read messages from mailboxes written in the following formats: USENET News, Berkeley/Unix, Babyl/RMAIL, and MH. New formats are straightforward to add.

PIEmail takes the mail from mailboxes and normalizes it into a mail folder format called Babyl, originally developed at MIT, and meant to provide a common intermediate format for a variety of mail systems. Babyl is also the format used by the default GNU Emacs mail reader, Rmail.

The `pm-babyl` class handles this mail format which consists of a folder header followed by a series of messages, with cached information about each message and its attributes preceding the normal message headers. The `pm-babyl` class encapsulates all of this specific formatting in order to provide other parts of PIEmail with structured access to message and folder contents.

Babyl folder methods are provided to test whether a folder is in Babyl format, to operate upon folder attributes, to create folders, to attach or detach mailboxes from a folder, to cache information about a folder's contents, and to read mail from mail drops or other folders.

Babyl message methods include extraction of named header fields, and conversion to a Babyl formatted message.

The `pm-attr` class supports Babyl message attributes.

The `pm-fdr` class supports PIEmail folder operations. The `pm-msgsumm` class is closely related as it manages summaries of the messages within folders. `Pm-fdr` may be thought of as a specialization of the `pm-set` class which manages ordered sets of messages.

The `pm-fdr` class is generic in the sense that its operations do not rely on the internal formats of the folders upon which it operates. Each folder's type is determined when the folder is opened by examining its data format. This allows PIEmail to dispatch to folder-type-specific operations whenever necessary, permitting it to read folders in a variety of formats such as UNIX BSD mail, Babyl, and MH folders. Thus, support for a new folder type can be added, basically by providing the same interface that the `pm-babyl` class does. `Pm-fdr` will utilize it without any modification. (In object-oriented parlance, we say that `pm-fdr` is a client of `pm-fdr-type` which has subclasses of `pm-babyl` and `pm-unix`.)

The organization of PIEmail data is a descending hierarchy of user, folder, subfolders and message. This hierarchy translates directly to a directory and file-based organization for PIEmail information. A subfolder's name is prefixed with the name of its parent folder and a '.' character. For example, piemail.design could hold messages on this design work with the parent folder, piemail, holding more general PIEmail-related messages.

Folder options include whether a summary should be produced whenever the folder is opened, whether the summary is recomputed after each folder change, and whether messages marked for deletion are automatically expunged whenever the folder is saved.

Pm-fdr-mode defines an interactive mode for operating on folders. It provides operations to create, open, close, delete, expunge, rename, filter, get new mail, save and summarize folders. Additionally, one can attach and detach attributes and mailboxes, test for the existence of a folder or any of its attributes.

`Pm-fdrsumm-mode` offers the same set of basic operations over folders that `pm-fdr-mode` does but in a form that lets one operate on multiple folders at once, since it lists folders and summarizes their states, one per line. By marking a set of folders and then applying operations, one can efficiently manage mail split across folders.

4.5 Messages

`Pm-msg` is a large class because it contains all of the non-attribute operations on messages. Messages may be composed, copied, deleted, edited, included, forwarded, jumped to, marked, merged into a digest, printed, replied to, saved, sent, undeleted, and split apart from a digest.

`Pm-msg` provides random access to messages by absolute message number (physical folder order) or by relative numbering (when a filter has been applied to a folder). Dynamic filtering can also be done during which a predicate is applied to each encountered message when a next or previous command is given and only messages meeting the predicate criteria are shown. The predicate may involve criteria that change as one walks through a set of messages so that the behavior will be different than if the set were pre-filtered.

Besides random access to messages, other movement facilities include movement to the first or last message within a set, to the beginning or end of a message (or its body), forward or backward a line, a page, or a windowful.

Expiration dates may be set on messages. These dates may then be used by message processing rules or filters. An expiration header is added to such messages to alert the user that messages are due to expire. When the expiration date arrives or passes and an expunge is done on the folder, expired messages are removed along with any deleted messages.

Message priorities may be set by name (e.g. urgent) or number (lower numbers mean higher priorities). Priorities can be used as a sort field when ordering messages.

It is common practice to delete messages after archiving or printing them. `Pm-msg` allows a user to attach other operations to any of its interactive commands. So one can specify that message print operation perform a message delete after if finishes just as easily as one can set the message to expire in a week. This allows for a high level of personalization at the message level.

4.5.1 Message Attributes

PIEmail messages have attached attributes which serve as properties that may be tested during rule or query processing and may also be inspected visually during interactive use of PIEmail. The standard message view mode line lists the attributes set for the current message.

Some attributes have binary values and are added and removed to indicate present state, such as deleted, answered, or filed. Others are multi-valued, such as priority.

Here is a summary of standard PIEmail message attributes. Others may be defined by users.

answered A reply has been mailed to the sender.

copied Message has been copied to another folder.

deleted Message is marked for deletion by the expunge operator.

edited Message has been edited in some way since reception.

expired Message is now due for time-based removal.

filed Message has been saved to a file.

forwarded Message has been resent to another address.

precious Messages with this attribute should never be deleted.

printed Message has been printed to paper from within PIEmail.

priority Message prioritization by keyword or number.

timely Message is marked for expiration on some date.

undigestified
A digest message has been unpacked into individual messages.

unseen Message has not yet been read.

The `pm-attr` class handles message attributes. It provides methods to set and get attributes and their values, to clear attributes, to toggle binary attributes and to test whether a message or message set contains an attribute. It also permits a user to define and attach his own attributes to messages.

4.5.2 Message Headers

Message header field operations are part of the `pm-hdr` class. It provides the low level, format-dependent interfacing to message headers. It can assist the `pm-msg` class by taking a list of header fields to hide from view and returning only other header fields for viewing. Header field values may be requested, header fields may be tested for existence, and fields may be added and deleted.

Priorities, for instance, are added to messages by adding a Priority header field with the priority's numeric value. Message expiration is also handled this way. An expiration date header is added to the message and checked for expiration whenever the folder is opened.

4.5.3 Message Sets

Message sets are common groupings of messages. All the messages following a particular thread of conversation and all the messages in a folder are some examples.

Most of the operations that can be applied to individual messages may also be applied to sets through the `pm-set` class. These include: copy, delete, display, expire, print, save, undelete, and attribute setting and testing. Additional set-specific operations include: filter, list, size, and sort.

4.5.4 Message Summaries

Message summaries provide an overview of message sets according to a selected view. **Pm-msgsumm** is the message summary class. **Pm-msgsumm-mode** supplies an interactive mode for message summaries. It offers the same facilities as **pm-msg-mode** and additionally permits selection of different views to obtain different perspectives over message sets.

4.6 Dates

PIEmail should eventually have very rich time and date handling support, but initially, the **pm-date** class offers simple support. Internet mail message date formats [*Crocker82*] are recognized and parsed into an internal format for comparison operations. This format also normalizes times and timezones so that message times from different parts of the world may be compared. Date ranges may also be supported so that actions can be scheduled to occur within a particular time window and rules can test such things as whether a mail message arrived within the first quarter of a year.

4.7 Personalization Facilities

4.7.1 Task-based Messaging

Most mail systems associate mail addresses with user identification names in one-to-one correspondence. This ignores the fact that people need to manage tasks that require them to wear different hats. Some task switches occur infrequently (over years) while others occur many times in a day or a week. The latter kind are not served by a system that leaves all mail for a user in a single mailbox.

Task addresses are user-defined slots whose names may be published so that mail on any subject sent to each address can be handled similarly. Task addresses offer a dynamically configurable means of organizing mail around personal work needs since they may be freely used within rule predicates. A single rule can file mail for each task into a separate folder.

Task addresses are created, deleted and managed by users rather than administrators. They allow users to specialize their mail address for any number of tasks, without permitting 'spoofing,' or imitation of other users, since each task address is a superset of a user's regular mail address. So for example, <stu@brown.edu> could use <stu.piemail@brown.edu> or <stu.piemail.ui@brown.edu> to discuss his work in the design of a PIEmail user interface.

Eventually, PIEmail will offer a standard means for requesting available task-based addresses and integrating them into rolodex records. A number of security issues remain to be worked out, since one generally wants to limit the distribution of task-oriented information to particular parties. This implies that one must be able to check somehow that the originating address of the request is valid and not an impersonation.

In anticipation of this, let us consider a few examples of task address usage.

Brent, who is a surveyor, a volunteer fireman and an avid runner may create addresses for three roles: surveyor, fireman and runner. His addresses would be: Brent.surveyor, Brent.fireman and Brent.runner to which anyone may mail. Of course, any desired breakdown may be used. Some people may prefer to assign roles for long and short-term work tasks.

To help people determine what addresses Brent has defined, assuming they know his base address is Brent, they can mail to Brent with a subject line of: Send PIEmail task addresses. A standard PIEmail rule processes such messages and sends a response listing Brent's available addresses, tailored specifically to the sender. Brent's address configuration file might look like this:

```
============================================================
    Task              -  Comment
      Addresses to which task is accessible
============================================================
*   Brent.surveyor    -  9am to 5pm,  Acme Surveying
      acme.com
*   Brent.fireman     -  8pm to 11pm, Akers County Fire
      akers.ohio.us
*   Brent.runner      -  mornings and weekends, major hobby
```

All lines after the first are used to indicate addresses to which this address is to be published. Here Brent has decided that only Acme employees should be told of and be able to mail to his surveyor role. In his fireman role, he has decided that only people with Akers county addresses be able to use it. And anyone may contact his runner role. His regular address of Brent, also remains open to all.

Task addresses provide two major benefits. They provide a pre-categorization of messages since messages sent to the same address are easily grouped together. Secondly, task addresses serve as attributes which may be tested during message rule evaluation in order to determine actions that should be taken with regard to a message. Brent may wish to file runner addressed messages to different folders based on any races that they mention. One rule attached to his runner address can do precisely this.

Take another example. Suppose a user, normally mailed to as Lars is head of an executive council meeting. He wants to deal with all correspondence regarding this task separately, so he creates a task address that assigns a corresponding mail box for Lars.exec-conf and informs potential conference attendees that all such correspondence should be addressed there. From that point on, he selects that address as the reply-to location whenever he mails to the group of executives, and he establishes a rule to catch errantly addressed mail (to his standard mail address) and to refile it appropriately.

After the conference when the task is no longer needed, he can remove the task address and the rule that redirects mail there. He can leave the task mail around so that he has access to the archive of conference messages or he can delete it together with the other task artifacts.

Finally, there is Megan who often must survey people on different topics. She uses a single program to extract survey responses and summarize resulting statistics. It would be useful for the system to recognize a survey message as such and to direct responses so that the response analysis program is run on each one as it arrives. The surveyor may then check the running tallies any time throughout the survey period. She can direct all survey responses to a task to simplify this automatic processing.

See Appendix D [Sendmail Tasks], page 81, if you use Sendmail as your mail transfer agent, for information on how to configure it for task address support.

4.7.2 Rules

Rules are the prime means of personalized automation within PIEmail. Rules specify conditional predicates that determine when they should be tested and in what contexts they should activate. Activation of a rule causes the evaluation of its action components. Rules have access to all folders and to entire messages. Rules may be applied to individual messages or message collections.

Rules can be used to analyze both incoming and outgoing messages in order to perform extensive sets of personalized actions. Possible actions include categorizing messages by topic and priority for reading, forwarding messages to local newsgroups or other interested parties and linking messages into public archives, as part of an organizational knowledge base. Rules should also simplify the creation of mail servers which deliver information automatically. Rules may be triggered when PIEmail is started, when new mail is read, when folders are opened or closed, when messages are archived or sent, and when PIEmail is quit.

Analysis of outgoing messages can be used not only to determine where to log such messages but also to influence the handling of responses. For example, one may want all responses filed into a single folder, in which case an appropriate task address can be generated and placed into the "Reply-to" field of the outgoing message, directing all replies to a personalized location.

The most common use of rules is to automate the folders into which messages are filed. Rules may use any header or body information to determine where to file a message and what priority to give it. Rule sets are built up in a recursive manner in which the output of the rules over a set of messages is used to refine that same rule logic. Although this process never ends, since one's interests and work tasks never wholly stop changing, we find that it converges quite quickly for any period of time. Within the course of a few days of usage, one can quickly build up quite sophisticated classification patterns which dramatically improve upon any classification that could be done manually.

Presently, rules must be specified programmatically, using a simple Lisp syntax like:

```
(if (pm-hdr-string subject "PIE")
    (pm-msg-file-to "pie"))
```

which says to file messages containing the term "PIE" in their Subject header fields to the 'pie' folder.

The rule language must be sufficiently flexible to permit growth in the system facilities and to allow succinct expression of common message handling policies and individual rules. (We do not concern ourselves with the security of this rule language since we assume that each user or responsible administrator has sole responsibility for each rule set and that any rule-based actions encoded in incoming messages may be inspected before activation, that is, they are never automatically activated.)

We expect to provide a form-based means of specifying rules. The form would name the rule, provide its evaluation context, predicate and actions. A number of predicates and actions will be standard with PIEmail so that they are quite easy to specify.

As an example, suppose Brent is involved in the Akers Open race upcoming in two weeks and he wants to file messages concerning that race into a folder named Open. The appropriate rule could be generated from a form filled out as follows:

```
-------------------------------------------------------
Rule: akers-open
From:
To: Brent.runner
CC:
Subject: Akers Open
Date:    Today + 2 weeks

Body:

Action:  file to Open
-------------------------------------------------------
```

A programmatic encoding of the same rule would be:

```
(pm-rule akers-open (task: runner)
  (and (pm-hdr-string subject "Akers Open")
       (pm-date-less-than (+ (pm-date-today) 14))
       (pm-msg-file-to "Open")))
```

4.7.3 Views

Views are a pervasive means of information display within PIEmail. The use of a view over an information base, rather than a fixed display format allows one to customize both display contents and display formats throughout PIEmail.

PIEmail views, forms, filters and sorters are closely related. Forms specify display layouts and formatting for a set of fields. Filters produce subsets from collections of information, keeping only those entities that match filter specifications. Sorters order messages within a set according to desired attributes.

Views utilize forms, filters and sorters to produce perspectives on entities, such as folders and messages. Collection views are especially common since they summarize related entities, helping one to take in the relationships that exist. The results of queries are always displayed as collection views. Filters may also be applied independently of view selection, so that one can affect the collection being viewed without altering its display format. Forms, views, filters and sorters may all be named like other PIEmail personalization facilities, to afford quick application over a set of informational entities.

For example, apply the messages-from-this-month view to all all mail that comes to my standard mail address. This would produce a folder summary view reporting only those messages that match the filter applied by the view, namely those received during the current month.

As you might expect, there are `pm-form`, `pm-view`, `pm-filter` and `pm-sorter` classes. PIEmail also provides more specialized types of the form and view classes to target the special needs of particular mail entities. `Pm-msgview` and `pm-summform` are two examples.

4.7.4 Priorities

Message priorities are assigned by recipients (or their rule sets) to provide an easy means to rank or filter messages for processing. Priorities are just like other multi-valued message attributes except that special operators are provided to make use of them.

Priorities are always integers but may be assigned names like 'urgent' for ease of use. Whenever a priority is requested, either the name or number may be given. Priority value ranges are user

selectable but default from one to ten, with lower numbers representing the more urgent priorities. Standard priority names in decreasing order are: urgent, high, medium, low, and trivial. *Pm-priority-alist* specifies the priority values and names available to PIEmail.

If a message is not assigned a priority by a rule or interactively by a user, it is given the default medium priority, indicating that it has not yet been judged important or unimportant. Because of this use, medium priorities are not shown in message summaries as are other priorities. The default priority can be changed by setting *pm-priority-default*.

4.7.5 Links

PIEmail includes a powerful associative linking feature, explained in much more detail in the next chapter. Links allow one to reference relevant material scattered across networks within messages, without the need to copy the material. This allows one to reference much more material than could be placed within a message. It minimizes the size of messages and permits the reader to view the most up to date version of the referenced information whenever desired.

Links are accessed via buttons embedded in messages. The PIEmail reader allows users to click on buttons to see what they do or to activate them. When link buttons are activated they traverse their links and display the associated referents.

Links are typed for a number of reasons. The code associated with each type need not be sent with each link or message. Both sender and receiver need only a single copy of the type which can process any number of its link instances. Types may be personalized to perform specially in local environments. A change to the type changes the behavior of all associated links, perhaps providing personalized views of information. New types can be quickly prototyped and delivered for use throughout organizations. Finally, types can offer rule-like condition-action behavior whereby a button activation equates to a rule evaluation.

Links allow one to build up informational contexts. Global buttons which can be activated by name at any time permit quick access to contexts, which themselves often contain more buttons. This allows one to link together various kinds of documents. Available contexts then are in easy reach when browsing or composing a message.

Context comes in both explicit forms, where a reference to another artifact is embbedded within the message, and implicit forms, where the receiver (or an agent of his) associates other artifacts with the message or conversation thread.

4.8 Performance

This is naturally a low priority while we are still developing the conceptual and operational framework for PIEmail. But we should put some thought into how we can keep performance degradation down as message set sizes grow. One idea is to store message headers in a database with multiple indices to speed the most common queries, e.g. from, to, time-based and subject lookups. Each record would have a pointer to its message body which usually would take significantly longer to retrieve than the header.

5 Hyperbole

5.1 PIEmail Integration

Present day mail systems include machine-readable references within mail message headers. For example:

```
In-Reply-To: Earl Johnson's message of
    Tue, 7 Apr 92 07:12:44 -0400 <9204071112.AA02966@delay.cs.brown.edu>

References: <9204091737.AA23973@flanger.cs.brown.edu>.
```

But there is no reason to limit the placement of references to mail headers, nor the references themselves to machine readable forms that point only to other mail messages.

PIEmail supports two new forms of references that permit easy inclusion of hypertextual references in e-mail, through its integration with our personalized information manager, Hyperbole. Such references may appear within any textual mail header or message body. The references themselves are human readable, with the machine-readable reference data hidden from view. The first type is an *explicit* reference whose label and referent are specified by the message sender during message creation. The second type is an *implicit* reference which is recognized according to context by using an expandable set of type definitions.

For example, suppose Lindsey's committee is drafting a Be All Your Parents Weren't recruiting specification for the army, she might define a new implicit reference type. The type could say that any pattern like 'see section 8' should refer to the appropriate section in the specification. She then could activate any such pattern as an implicit hypertext button from her PIEmail reader in order to display the referenced section.

5.2 Hyperbole Concept

Hyperbole is an open, efficient, programmable prototype information management system with a hypertextual flavor that we have developed to explore PIE concepts. Its integration with PIEmail, GNU Emacs, and Epoch offers a rich working environment that can be tailored to a wide variety of information handling needs. It is simple to use since it introduces only a few new mechanisms and provides user-level facilities through a menu interface, that may be controlled from either a mouse or keyboard. Since Hyperbole is more mature than PIEmail, we describe it operation in greater detail throughout this chapter.

Hyperbole consists of three parts:

1. an interactive information management interface which anyone can use;
2. a set of programming library classes for system developers who want to integrate Hyperbole with another user interface or as a back-end to a distinct system;
3. a set of hyper-button types that provides core hypertext and other behaviors; users can make simple changes to button types and those familiar with Emacs Lisp can quickly prototype and deliver new types.

A Hyperbole user works with *buttons*; he may create, modify, move or delete buttons. Each button performs a specific action, such as display of a file or execution of a shell command.

Presently, there are two categories of Hyperbole buttons: *explicit buttons*, the ones that Hyperbole creates, and *implicit buttons*, buttons created and managed by other programs or embedded within the structure of a document and recognized contextually by Hyperbole. Explicit Hyperbole buttons may be embedded within any type of text file or message. Implicit buttons may be recognized anywhere within text also, depending on the implicit button types that are available.

Hyperbole buttons may be clicked upon with a mouse to activate them or to describe their actions. Thus, a user can always check what a button will do before activating it.

Hyperbole does not enforce any particular hypertext or information management model, but instead allows you to organize your information in large or small chunks as you see fit. Some of its most important features include:

- Buttons may link to information or may execute procedures, such as starting or communicating with external programs;
- Buttons may be embedded in any textual portion of electronic mail messages;
- Other hypertext and information retrieval systems may be encapsulated under a Hyperbole user interface. PIEmail uses the mouse handlers built into Hyperbole to offer a consistent interface. All of Hyperbole's features described in this chapter are available to PIEmail.

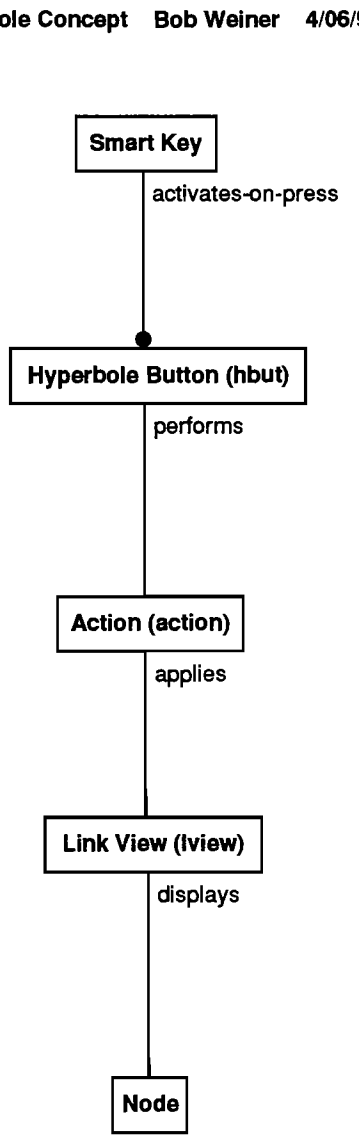
Typical Hyperbole uses include:

personal information management
: Overlapping link paths provide a variety of views into an information space. A search facility locates buttons in context and permits quick selection.

documentation browsing
: Cross-references may be embedded within documentation. One can add a point-and-click interface to existing documentation, link code with associated design documents, or jump to the definition of an identifier from its use within code or its reference within documentation. A table of contents composed from buttons can then be mailed to a group of potential readers, allowing them to quickly check into sections of interest with little interruption to their workflow.

help/training systems
: Tutorials with embedded buttons can show students how things work while explaining the concepts, e.g. an introduction to local commands. This technique can be much more effective than descriptions alone. Hyperbole comes with a demonstration of its features that utilizes this principle. Teaching assistants can answer questions electronically and provide pointers that reference relevant material.

archive managers
: Programs that manage archives from incoming information streams may be supplemented by having them add topic-based buttons that link to the archive holdings. Users can then search and create their own links to archive entries. For example, summaries of incoming messages could be stored for efficient searching, each with a link to the associated message.



You can see the simplicity of the basic Hyperbole concept and the linearity of its control flow from this diagram. A press of a Smart Key (see Section 5.5.1 [Smart Keys], page 38) over a Hyperbole button triggers the button's action which displays a view of some information artifact. Keep that in mind as we explore Hyperbole's design in more detail. Much of the technical complexity that we expose throughout this chapter is hidden from the user, who lives in a point-and-click world.

5.3 Button Concepts

Hyperbole buttons persist across Emacs sessions, so they provide a convenient means of linking from one information source to another.

5.3.1 Explicit Buttons

Hyperbole creates and manages *explicit buttons* which look like this `<(fake button)>` to a Hyperbole user. They are quickly recognizable, yet relatively non-distracting as one scans the text in which they are embedded. The text between the `<(` and `)>` delimiters is called the *button label*. Spacing between words within a button label is irrelevant to Hyperbole, so button labels may wrap across several lines without causing a problem. See Section 5.3.6 [Label Keys], page 34. When Hyperbole is run under Epoch, it can automatically highlight any explicit buttons in a buffer and make them flash when selected.

Hyperbole stores the *button data* that gives an explicit button its behavior, separately from the button label, in a file named '`.hypb`' within the same directory as the file in which the button is created. Button data is comprised of individual *button attribute* values. A user never sees this data in its raw form but may see a formatted version by asking for help on a button.

Explicit buttons may be freely moved about within the buffer in which they are created. (No present support exists for moving buttons between buffers, but this is a high priority item for development.) A single button may also appear multiple times within the same buffer.

Explicit buttons are associated with action types which determine the actions they perform. *Link action types* connect buttons to particular types of referents. *Activation* of such buttons then displays the referents.

Hyperbole does not manage referent data; this is left to the applications that generate the data. This means that Hyperbole provides in-place linking and does not require reformatting of data to integrate it within a Hyperbole framework.

5.3.2 Action Types and Actions

Action types provide action procedures that specify button behavior. The use of action types provides a convenient way of specifying button behavior without the need to know how to program. New forms of explicit buttons may be created simply by adding new action types to a Hyperbole environment. The arguments needed by an action type are prompted for at button creation time. When a button is activated, the stored arguments are fed to the action type's *action* body to achieve the desired result. Hyperbole handles all of this transparently.

Standard action types are:

`annot-bib`
: Follows internal ref KEY within an annotated bibliography, delimiters=[].

`eval-elisp`
: Evaluates a Lisp expression LISP-EXPR.

`exec-kbd-macro`
: Executes KBD-MACRO REPEAT-COUNT times.

`exec-shell-cmd`
: Executes a SHELL-CMD string asynchronously.

`hyp-source`
: Displays a buffer or file from a line beginning with 'hbut:source-prefix'.

`kbd-key` Executes the function binding for KEY-SEQUENCE, delimited by {}.

`link-to-Info-node`
: Displays an Info NODE in another window.

`link-to-buffer-tmp`
: Displays a BUFFER in another window.

`link-to-directory`
: Displays a DIRECTORY in Dired mode in another window.

`link-to-ebut`
: Performs action given by another button, specified by KEY and KEY-FILE.

`link-to-file`
: Displays a PATH in another window scrolled to optional POINT.

`link-to-file-line`
: Displays a PATH in another window scrolled to LINE-NUM.

`link-to-mail`
: Displays mail msg with MAIL-MSG-ID from MAIL-FILE in other window.

`link-to-regexp-match`
: Finds REGEXP's Nth occurrence in FILE and displays location at window top.

`link-to-string-match`
: Finds STRING's Nth occurrence in FILE and displays location at window top.

`man-show` Displays man page on TOPIC, which may be of the form `<command>(<section>)`.

`rfc-toc` Computes and displays summary of an Internet rfc in BUF-NAME.

5.3.3 Global Buttons

Access to explicit buttons depends upon the information on your screen because you have to be able to point to a button label to activate it. Sometimes it is useful to activate particular buttons without regard to the particular information with which you are presently working. In such instances, you use *global buttons*, which are simply explicit buttons that may be activated or otherwise operated upon by label name. If you want a permanent link to a file section that you can follow at any time, you can use a global button. Or what about a watch-what-I-do macro that you use frequently? Create an `exec-kbd-macro` button with an easy to type name and then you can activate it whenever the need arises.

5.3.4 Implicit Buttons and Types

Implicit buttons are those defined by the natural structure of a document. They are identified by contextual patterns which limit the locations or states in which they can appear. Their behavior is determined by one or more actions which they trigger, where an action is derived from either a Hyperbole action type specification or an Emacs Lisp function. Implicit button types may use the same action types that explicit buttons do.

Implicit buttons never have any button data associated with them. They are recognized in context based on predicate matches defined within implicit button types. For example, Hyperbole recognizes file names enclosed in double quotes and can quickly display their associated files in response to simple mouse clicks.

Standard implicit button types include:

`annot-bib`
: Displays annotated bibliography entries referenced internally, delimeters = [].

`dir-summary`
: Detects filename buttons in files named "MANIFEST" or "DIR".

`grep-msg` Jumps to line associated with grep or compilation error msgs. Messages are recognized in any buffer.

`hyp-source`
: Makes source entries in Hyperbole reports into buttons that jump to source.

`Info-node`
: Makes "(file)node" buttons display the associated Info node.

`kbd-key` Executes a key sequence delimited by curly braces.

`man-apropos`
: Makes man apropos entries display associated man pages when selected.

`pathname` Makes a delimited, valid pathname display the path entry. Also works for delimited and non-delimited ange-ftp pathnames.

`rfc-toc` Summarizes contents of an Internet rfc from anywhere within rfc buffer.

5.3.5 Button Label Keys

Hyperbole uses a normalized form of button labels called button keys (or label keys) for all internal operations. The normalized form permits Hyperbole to recognize buttons that are the same but whose labels appear different from one another, due to text formatting conventions. As an illustration, all of the following would be recognized as the same button.

```
<(fake button)>

<( fake      button)>

<(
   fake button
)>

Pam>  <(fake
Pam>    button)>

;; <(fake
;;   button)>

/* <( fake      */
/*    button )> */
```

The last three examples demonstrate how Hyperbole ignores common fill prefix patterns that happen to fall within the middle of a button label that spans multiple lines. As long as such buttons are selected with point at a location within the label's first line, the button will be recognized. The variable *hbut:fill-prefix-regexps* holds the list of fill prefixes recognized when embedded within button labels. All such prefixes are recognized (one per button label), regardless of the setting of the GNU Emacs variable, *fill-prefix*, so no user intervention is required.

5.3.6 Operational and Storage Formats

Hyperbole uses a terse format to store explicit buttons and a more meaningful one to show

users and to manipulate during editing. The terse format consists solely of button attribute values whereas the edit format includes an attribute name with each attribute value. A button in edit format consists of a Lisp symbol together with its property list which holds the attribute names and values. In this way, buttons may be passed along from function to function simply by passing the symbol to which the button is attached. Most functions utilize the pre-defined *hbut:current* symbol by default to store and retrieve the last encountered button in edit format.

The `hbdata` class handles the terse, stored format. The `hbut`, `ebut`, `gbut` and `ibut` classes work with the name/value format. This separation permits the wholesale replacement of the storage manager with another, with any interface changes hidden from any Hyperbole client programming.

5.3.7 Programmatic Button Creation

A common need when developing with Hyperbole is the ability to create or modify explicit buttons without user interaction. For example, an application might require the addition of an explicit summary button to a file for each new mail message a user reads that contains a set of keywords. The user could then check the summary file and jump to desired messages quickly.

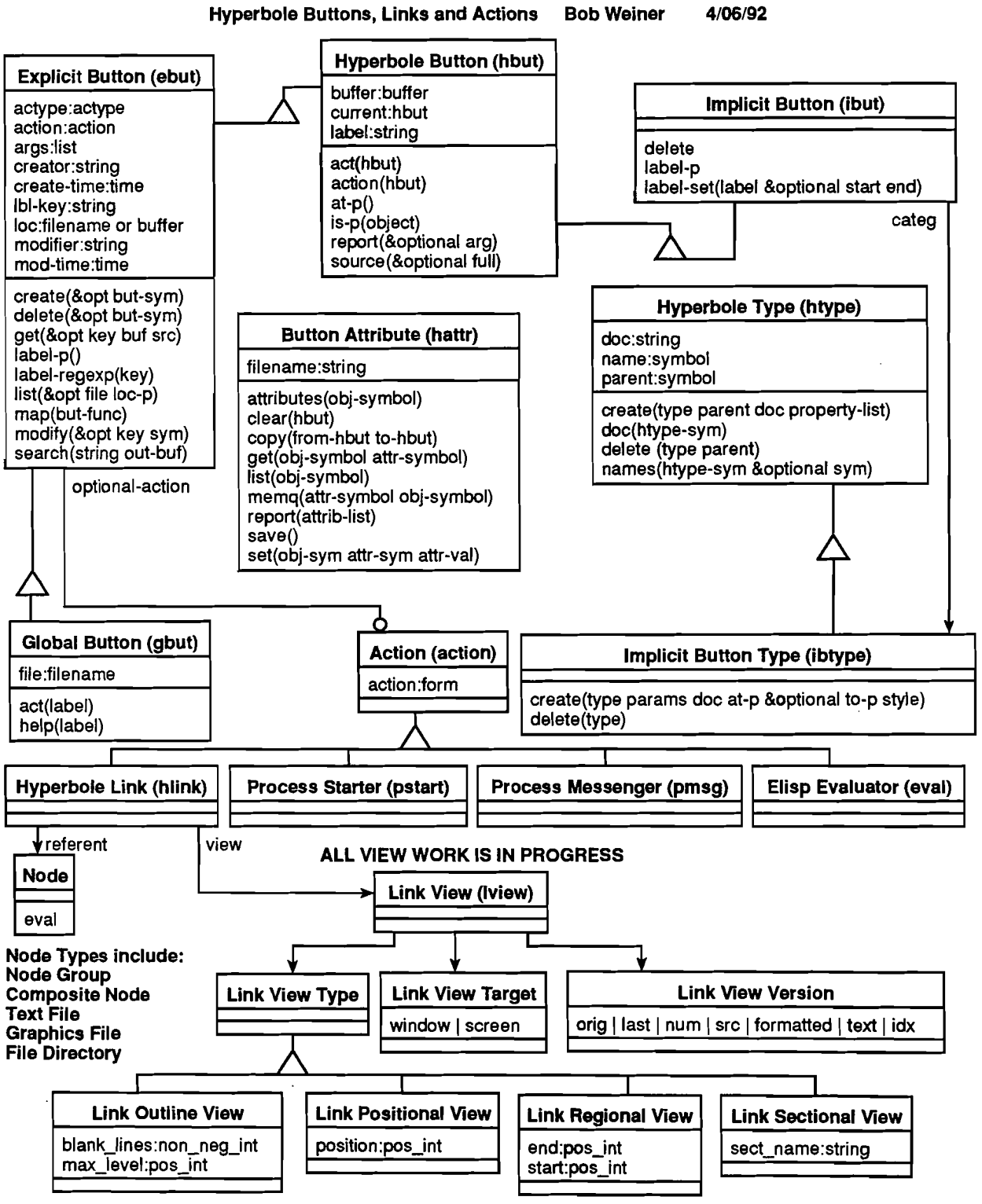
The Hyperbole `ebut` class supports programmatic access to explicit buttons. The documentation for `(ebut:create)` explains the set of attributes settings necessary to create an explicit button. For operations over the whole set of buttons within the visible (non-narrowed) portion of a buffer, use the `(ebut:map)` function.

5.4 Button and Type Precedences

Explicit buttons always take precedence over implicit buttons. Thus, if a button selection is made which falls within both an explicit and implicit button, only the explicit button will be selected. Explicit button labels are not allowed to overlap; Hyperbole's behavior in such cases is undefined.

If there is no explicit button at point during a selection request, then each implicit button type predicate is tested in turn until one returns non-nil or all are exhausted. The types are used in **reverse** order of definition, so that personal types defined after standard system types take precedence. It is important to keep this order in mind when defining new implicit button types. By making their match predicates as specific as possible, one can minimize any overlapping of implicit button type domains (those contexts in which a type predicates match).

Once a type name is defined, its precedence relative to other types remains the same even if you redefine the body of the type, as long as you don't change its name. This allows incremental modifications to types without having to worry about shifts in type precedence. Implicit button types can be listed in decreasing order of precedence by using the Hyperbole Types menu mentioned later.

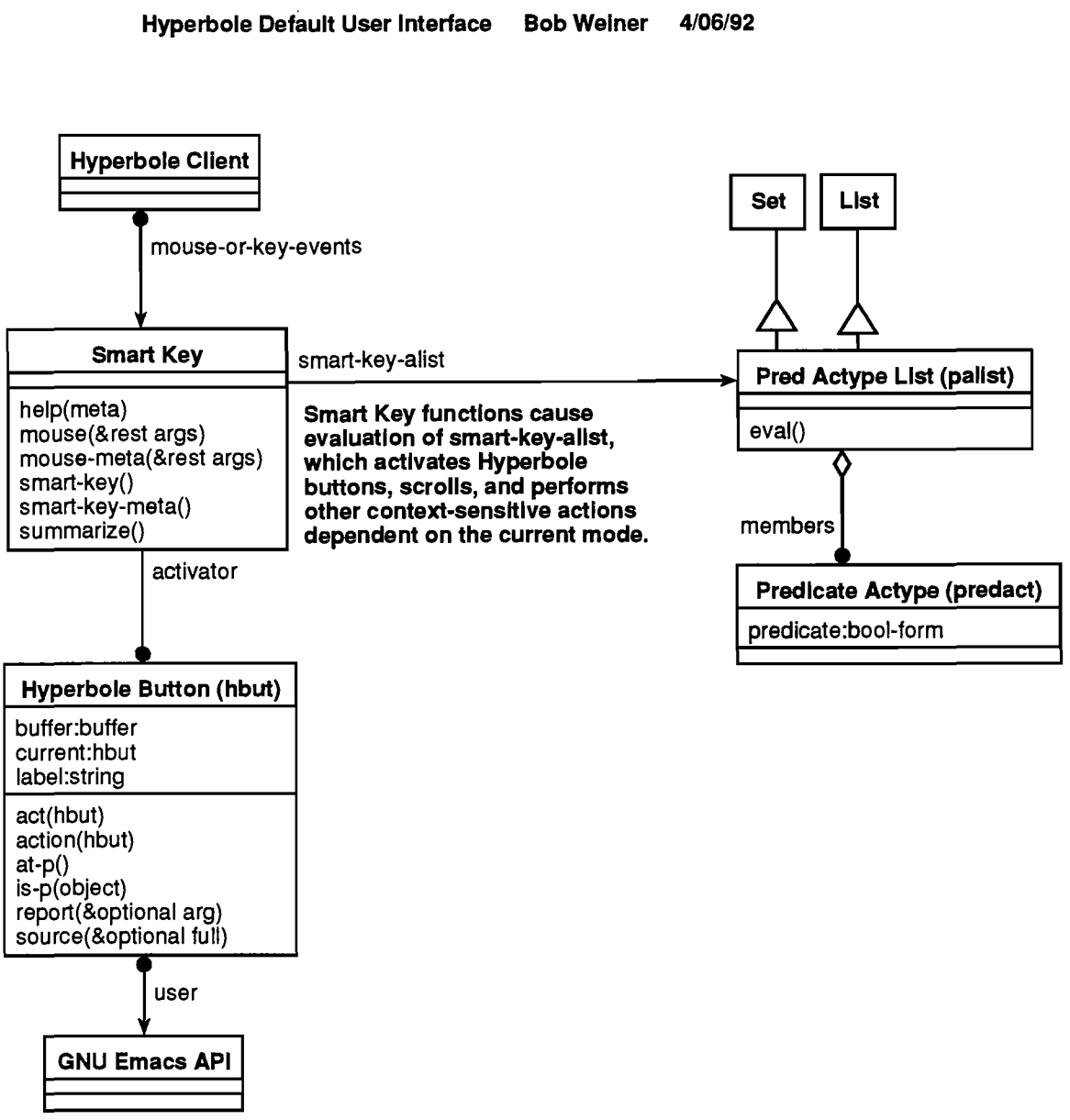

Hyperbole Buttons, Links and Actions Bob Weiner 4/06/92
Explicit Button (ebut)
actype:actype
action:action
args:list
creator:string
create-time:time
lbl-key:string
loc:filename or buffer
modifier:string
mod-time:time
create(&opt but-sym)
delete(&opt but-sym)
get(&opt key buf src)
label-p()
label-regexp(key)
list(&opt file loc-p)
map(but-func)
modify(&opt key sym)
search(string out-buf)
Hyperbole Button (hbut)
buffer:buffer
current:hbut
label:string
act(hbut)
action(hbut)
at-p()
is-p(object)
report(&optional arg)
source(&optional full)
Implicit Button (ibut)
delete
label-p
label-set(label &optional start end)
categ
Button Attribute (hattr)
filename:string
attributes(obj-symbol)
clear(hbut)
copy(from-hbut to-hbut)
get(obj-symbol attr-symbol)
list(obj-symbol)
memq(attr-symbol obj-symbol)
report(attrib-list)
save()
set(obj-sym attr-sym attr-val)
Hyperbole Type (htype)
doc:string
name:symbol
parent:symbol
create(type parent doc property-list)
doc(htype-sym)
delete (type parent)
names(htype-sym &optional sym)
optional-action
Global Button (gbut)
file:filename
act(label)
help(label)
Action (action)
action:form
Implicit Button Type (ibtype)
create(type params doc at-p &optional to-p style)
delete(type)
Hyperbole Link (hlink)
Process Starter (pstart)
Process Messenger (pmsg)
Elisp Evaluator (eval)
referent
view
ALL VIEW WORK IS IN PROGRESS
Node
eval
Link View (lview)
Node Types include:
Node Group
Composite Node
Text File
Graphics File
File Directory
Link View Type
Link View Target
window | screen
Link View Version
orig | last | num | src | formatted | text | idx
Link Outline View
blank_lines:non_neg_int
max_level:pos_int
Link Positional View
position:pos_int
Link Regional View
end:pos_int
start:pos_int
Link Sectional View
sect_name:string

5.5 Using Hyperbole

This section covers more of the user view of Hyperbole. Often a number of overlapping interaction methods are provided to support different work styles and hardware limitations. A user need learn only one with which he can become comfortable, in such instances.

The 'DEMO' file included in the Hyperbole distribution demonstrates many of Hyperbole's standard facilities. It provides a good example of how buttons may be used. It is replicated as an appendix for reference. See Section C.6 [Hyperbole Demo], page 78.

5.5.1 Smart Keys



Hyperbole provides two special keys that perform different operations in different contexts, as explained below. By default, these two keys are bound to your middle and right mouse keys when Hyperbole is run under an external window system (mouse configuration is automatic for the X Window System, SunView, and Apollo's Display Manager if your GNU Emacs program has been built with support for any of these window systems).

The middle key is called the *primary Smart Key* (or just Smart Key) and the right is called the *secondary Smart Key* (or Smart Meta Key). By default (if *smart-key-init* is left set equal to `t` in

'`hsite.el`'), then `{C-u M-RET}` may also be used as the primary Smart Key and `{M-RET}` may be used as the secondary Smart Key. These key bindings allow context sensitive operation from any keyboard.

The primary Smart Key generally selects entities and activates buttons. The secondary Smart Key generally provides help, such as reporting on a button's attributes, or serves a complementary function to whatever the primary Smart Key does within a particular context.

When Hyperbole is installed, a key may be bound which allows you to switch between the Smart Key mouse bindings and your prior ones. `C-h w sm-mouse-toggle-bindings RTN` shows any key which performs this command.

A prime design criterion of Hyperbole's user-interface is that one should be able check what an operation will do before applying the operation. Thus, one can get help on buttons and menu items before activating them by using the secondary Smart Key. When you use a mouse and you want to find out what either of the Smart Keys does within a context, depress the one you want to check on and hold it down, then press the other and release as you please. A help buffer will pop up explaining the actions that will be performed in that context, if any. A press of the primary Smart Key at the end of that help buffer will restore your display to its configuration prior to invoking help.

You can also get a complete listing of what the Smart Keys do in different contexts by pressing the secondary Smart Key within a context that has no special Smart Key function, e.g. the middle of a paragraph. The following table is an example of what you will be shown. Much of the browsing power of Hyperbole comes from use of the Smart Keys, so it is worth spending some time examining the table. It may appear daunting at first, but as you practice and notice that the Smart Keys do just a few custom things per editor mode, you will find it easy to just point and click and let Hyperbole do the rest.

Context	Smart Key	
	Primary	Secondary
Hyperbole		
On a menu item	Item is activated	Item help
On an explicit button	Button is activated	Button help
Reading argument		
1st press at an arg value	Value copied to minibuffer	<- same
2nd press at an arg value	Value used as argument	<- same
In minibuffer	Minibuf arg is applied	Completion help
On an implicit button	Button is activated	Button help
Wrolo Match Buffer	Toggles narrow to one entry and edits entries	
Screen Control		
Line end, not end of buffer	Scrolls up a windowful	Scrolls down
End of Any Help buffer	Screen restored to previous state	
Special Modes		
C Mode	Jumps to id def	Jumps to next def
Any Lisp mode	Jumps to id def	Jumps to next def
Occur match	Jumps to match source line	<- same
Multi-buffer occur match	Jumps to match source line	<- same
Outline Major/Minor Modes	Collapses, expands, and moves outline entries	
Man Apropos	Displays man page entry	<- same
Man Pages	Follows cross refs, file refs and C code refs	
Buffer Menu	Saves, deletes and displays buffers	
Emacs Info Reader		
Menu Entry or Cross Ref	Jumps to referent	<- same
Up, Next or Prev Header	Jumps to referent	Jumps to prior node
File entry of Header	Jumps to top node	Jumps to (DIR) node
End of current node	Jumps to next node	Jumps to prev node
Anywhere else	Scrolls up a windowful	Scrolls down a wind
Subsystems		
Calendar	Scrolls or shows appts	Scrolls or marks dates
Dired Mode	Views and deletes files from directory listing	
GNUS News Reader	Toggles group subscriptions, gets new news, and browses articles	
Mail reader and Summaries	Browses, deletes and expunges messages	
Any other context (defaults)	Hyperbole top menu	Smart Key summary

You can see that button and mail handling are just a few of the things for which the Smart Keys are useful. They are designed to offer a convenient means of browsing throughout an environment without a need to remember any complicated modifier keys to press in conjunction with the mouse keys. Instead you point at the entity you want to operate upon and natural operations are performed. If you want to customize what happens when an entity is selected, you can define new implicit button types that override the default Smart Key behaviors.

5.5.2 Operating Menus

Hyperbole menus provide access to its user-level commands. Only one key binding is used for the top-level menu, but menu items may be selected from either the keyboard or via mouse clicks. When used with the keyboard, they provide rapid command access similar to regular key bindings.

The Buttons menu is invoked from the PIEmail Construct menu, from a key (by default, `{C-h h}`), or via a Smart Key press in a random context. The menu will appear in the minibuffer and should look about like so:

```
Hypb> Act ButFile/ Doc/ EBut/ GBut/ Hist IBut/ Msg/ Rolo/ Types/
```

All button items are selected via the first character of their names (letter case does not matter) or via a press of the Smart Key. "/" at the end of an item name indicates that it brings up a sub-menu. You can browse around the menus by hitting `{C-t}` to return to the top menu or `{q}` when you want to quit instead of selecting an item.

The above menu items can be summarized as follows:

Act Perform the action associated with any button at point.

ButFile/ Display a local or global file of buttons, providing easy access. The 'HYPB' file in the current directory for a local button file and '`~/.hyperb/HYPB`' for your global file. These are good places to start your button creation testing. See Section 5.5.4.6 [Button Files], page 43.

EBut/ All explicit button commands.

Doc/ Hyperbole documentation quick access.

GBut/ All global button commands. Global buttons are accessed by name rather than by direct selection.

Hist Jumps back to last position in button traversal history.

IBut/ All implicit button commands.

Msg/ Hyperbole-specific mail and news messaging support commands. This is used to send mail to the Hyperbole discussion list or to add/modify a personal entry on a Hyperbole mail list.

Rolo/ Hierarchical, multi-file rolodex lookup and edit commands. See Section 2.8 [Rolodex Use], page 12.

Types/ Documentation on Hyperbole types.

5.5.3 Entering Arguments

Many Hyperbole commands prompt you for arguments. The standard Hyperbole user interface has an extensive core of argument types that it recognizes. Whenever Hyperbole is prompting you for an argument, it knows the type that it needs and provides some error checking to help you get it right. More importantly, it allows you to click within an entity that you want to use as an argument and it will grab the appropriate thing and show it to you at the input prompt within the minibuffer. If you click on the same thing again, it accepts the entity as the argument and moves on. Thus, a double click registers a desired argument. Double-quoted strings, pathnames, mail messages, Info nodes, dired listings, buffers, numbers, completion items and so forth are all recognized at appropriate times. All of the argument types mentioned in the documentation for the Emacs Lisp `(interactive)` function are recognized. This direct selection technique is very easy to get used to with just a little practice.

Wherever possible, standard Emacs completion is offered, see section "Completion" in *the Gnu Emacs Manual*. Remember to use `{?}` to see what your possibilities for an argument are. Once you have a list of possible completions on screen, you can double click the Smart Key on any one to enter it as the argument.

5.5.4 Working with Explicit Buttons

Explicit buttons provide the building blocks for creating personal or organizational hypertext networks with Hyperbole. This section summarizes the user-level operations available for managing these buttons.

5.5.4.1 Creating

The most efficient way to create an explicit button interactively is to mark a short region of text in any fashion allowed by GNU Emacs and then to select the Hyperbole menu item sequence, Ebut/Create. You will be prompted for the button's label with the marked region as the default. If you accept the default and enter the rest of the information you are prompted for, the button will be created within the current buffer and Hyperbole will surround the marked region with explicit button delimiters to indicate success. The process becomes quite simple with a little practice.

If you do not mark a region before invoking the button create command, you will be prompted for both a label and a source buffer for the button and the delimited label text will be inserted into the chosen buffer after a successful button creation.

If a previous button with the same label exists in the same buffer, Hyperbole will add an *instance number* to the label when it adds the delimiters so that the name is unique. Thus, you don't have to worry about accidental button name conflicts. If you want the same button to appear in multiple places within the buffer, just enter the label again and delimit it yourself. Hyperbole will interpret all occurrences of the same delimited label within a buffer as the same button.

After Hyperbole has the button label and its source buffer, it will prompt you for an action type for the button. Use the `{?}` completion key to see the available types. The type selected determines any following values for which you will be prompted.

When creating link buttons, the best technique is to place the buffer in which you want your button and the entity to link to on screen at the same time. Mark the region of text to use for your button label, invoke the button create command from the menu, choose an action type which begins with `link-to-` and then use the direct selection techniques mentioned in Section 5.5.3 [Entering Arguments], page 41, to select the link referent. This becomes quite rapid after a few tries so that one can quickly create links from almost anywhere to anywhere else.

5.5.4.2 Renaming

Once an explicit button has been created, its label text must be treated specially. Any inter-word spacing within the label may be freely changed, as may happen when a paragraph is refilled. But a special command must be invoked to rename it.

The rename command operates in two different ways. If point is within a button label when it is invoked, it will tell you to edit the button label and then invoke the rename command again. The second invocation will actually rename the button. If instead the command is originally invoked outside of any explicit button, it will prompt for the button label to replace and the label to replace it with and then will perform the rename.

The rename command may be invoked from the Hyperbole menu via Ebut/Rename. A faster method is to use a key bound to the `hui:ebut-rename` command. Your site installation may include such a key. `{C-h w hui:ebut-rename RTN}` should show you any key it is on.

5.5.4.3 Deletion

Ebut/Delete works similarly to the Rename command but deletes the selected button. The button's delimiters are removed to confirm the delete. Presently there is no way to recover a deleted button; it must be recreated, but this will be remedied in the future.

5.5.4.4 Modification

Ebut/Modify prompts you with each of the elements from the button's data list and allows you to modify each in turn.

5.5.4.5 Location

The Ebut/Help menu can be used to summarize all of the explicit buttons within a single buffer. The buttons may then be activated directly from the summary.

Ebut/Search prompts for a search pattern and searches across all the locations in which you have previously created explicit buttons. It asks you whether to match to any part of a button label or only complete labels. It then displays a list of button matches with a single line of surrounding context from their sources. Any button in the match list may be activated as usual. A Smart Key press on the surrounding context jumps to the associated source line or a press on the filename preceding the matches jumps to the file without selecting a particular line.

There are presently no user-level facilities for globally locating buttons created by others or for searching on particular button attributes.

5.5.4.6 Button Files

It is often convenient to create lists of buttons that can be used as menus that offer centralized access to distributed information pools. These files can serve as useful roadmaps to help efficiently guide a user through both unfamiliar and highly familiar information spaces. Files that are created specifically for this purpose, we call *button files.*

The Hyperbole menu system gives quick access to two types of these button files: personal and directory-specific, through the ButFile menu. (The variable, *hbmap:filename*, contains the base name of these standard button files. Its standard value is '`HYPB`'.)

A personal button file may serve as a user's own roadmap to frequently used resources. Selection of the ButFile/PersonalFile menu item displays this file for editing. The default personal button file is stored within the directory given by the *hbmap:dir-user* variable whose standard value is '`~/.hyperb`'. The standard Hyperbole configuration also appends all global buttons to the end of this file, one per line, as they are created. So you can edit or annotate them within the file.

A directory-specific button file may exist for each file system directory. Such files are useful for explaining the contents of directories and pointing readers to particular highlights within the directories. Selection of the ButFile/DirFile menu item displays the button file for the current directory; this provides an easy means of updating this file when working on a file within the same

directory. If you want to view some other directory-specific button file, simply use the normal Emacs file finding commands.

. One might suggest that menu quick access be provided for group-specific and site-specific button files. Instead, link buttons to such things should be placed at the top of your personal button file. This provides a more flexible means of quick access.

5.5.4.7 Buttons in Mail

Hyperbole allows the embedding of buttons within electronic mail messages that are composed in Emacs with the standard (`mail`) command, normally bound to {`C-x m`} or with other Emacs-based mail composing functions. An enhanced mail reader like PIEmail can then be used to activate the buttons within messages just like any other buttons.

Hyperbole automatically supports the PIEmail, Rmail, see section "Rmail" in *the GNU Emacs Manual*, VM, see section "Top" in *the VM Manual*, and MH-e mail readers. Button inclusion and activation within USENET news articles is also supported in the same fashion via the GNUS news reader, see section "Top" in *the GNUS Manual*, if available at your site.

All explicit buttons to be mailed must be created within the outgoing message buffer. There is no present support for including text from other buffers or files which contain explicit buttons, except for the ability to yank the contents of a message being replied to, together with all of its buttons, via the (`mail-yank-original`) command bound to {`C-c C-y`}. From a user's perspective, buttons are created in precisely the same way as in any other buffer. They also appear just like any other buttons to both the message sender and the reader who uses the Hyperbole enhanced readers. Button operation may be tested any time before a message is sent. A person who does not use Hyperbole enhanced mail readers can still send messages with embedded buttons since mail composing is independent of any mail reader choice.

Hyperbole buttons embedded within received mail messages act just like any other buttons. The mail does not contain any of the action type definitions used by the buttons, so the receiver must have these or she will receive an error when she activates the buttons. Buttons which appear in message *Subject* lines are copied to summary buffers whenever such summaries are generated. Thus, they may be activated from either the message or summary buffers.

Nothing bad will happen if a mail message with explicit buttons is sent to a non-Hyperbole user. The user will simply see the text of the message followed by a series of lines of button data at its end. Hyperbole mail users never see this data in its raw form.

In order to alert readers of your mail messages that you can utilize Hyperbole mail buttons, the system automatically inserts a comment into each mail message that you compose to announce this fact. The variable, *smail:comment* controls this behavior. See its documentation for technical details. By default, it produces a message of the form:

Comments: Hyperbole mail buttons accepted, vX.XX.

where the X's indicate your Hyperbole version number. You can cut this out of particular messages before you send them. If you don't want any message at all, add the following to your '`~/.emacs`' file before the point at which you load Hyperbole.

```
(setq smail:comment nil)
```

A final mail-related facility provided by Hyperbole is the ability to save a pointer to a received mail message by creating an explicit button with a `link-to-mail` action type. When prompted for the mail message to link to, if you press the Smart Key on a message, the appropriate parameter will be copied to the argument prompt, as described in Section 5.5.3 [Entering Arguments], page 41.

5.5.4.8 Buttons in News

Explicit buttons may be embedded within outgoing USENET news articles and may be activated from news articles that are being read. This support is available for the GNUS news reader.

All Hyperbole support should work just as it does when reading or sending mail. See Section 5.5.4.7 [Buttons in Mail], page 44. When reading news, buttons which appear in message *Subject* lines may be activated within the GNUS subject buffer as well as the article buffer. When posting news, the *post-news* buffer is used for outgoing news articles rather than the *mail* buffer.

Remember that the articles you post do not contain the action type definitions used by the buttons, so the receiver must have these or she will receive an error when he activates the buttons. You should also keep in mind that most USENET readers will not be using Hyperbole, so if they receive a news article containing explicit buttons, they will wonder what the button data at the end of the message is. You should therefore limit distribution of such messages. For example, if most people at your site read news with GNUS and use Hyperbole, it would be reasonable to embed buttons in postings to local newsgroups.

In order to alert readers of your postings that you can utilize Hyperbole mail buttons embedded within personal replies, the system automatically inserts the same comment that is included within mail messages to announce this fact. See Section 5.5.4.7 [Buttons in Mail], page 44, for details and an explanation of how to turn this feature off.

5.6 Hyperbole Glossary

Concepts pertinent to operational usage of Hyperbole are defined here. There are enough of them that we have separated them from the PIEmail glossary included as an appendix, see Appendix A [Glossary], page 71.

`action` An executable behavior associated with a Hyperbole button. A specific class of actions which display entities are called *links*, such as a link to a file.

`action type`
A behavioral specification for use within Hyperbole buttons. Action types usually contain a set of parameters which must be given values for each button with which they are associated. An action type together with a set of values, called arguments, may be considered an *action*. *Actype* is a synonym for action type.

`activation`
Request for a Hyperbole button to perform its action. Ordinarily the user presses a key which selects and activates a button.

`ange-ftp` A non-standard GNU Emacs Lisp package which allows one to use pathnames that are accessible via the Internet File Transfer Protocol (ftp) just like other pathnames, for example when finding a file.

`argument` A button-specific value fed to a Hyperbole type specification when the button is activated.

`attributes`
Slot names associated with Hyperbole buttons. An *attribute value* is associated with each button attribute.

`button` A selectable Hyperbole construct which performs an action. A button consists of a set of attributes that includes: a textual label, a category, a type and zero or more

arguments. *Explicit buttons* also have creator, create time, last modifier, and last modifier time attributes.

Buttons provide the user's gateway to information. The user sees and interacts with button labels, the rest of the button data is managed invisibly by Hyperbole and displayed only in response to user queries.

`button activation`
: See *activation*.

`button attributes`
: See *attributes*.

`button data`
: Lists of button attribute values explicitly saved and managed by Hyperbole. One list for each button created by Hyperbole.

`button file, local`
: A per-directory file named 'HYPB' that may be used to store any desired buttons and may then be displayed via a menu selection whenever a user is within that directory.

`button file, personal`
: A per-user file named 'HYPB' that may be used to store any desired buttons and may then be displayed via a menu selection.

`button key`
: A normalized form of a *button label* used internally by Hyperbole.

`button label`
: A text string that visually indicates a Hyperbole button location and provides it with a name and unique identifier. Within a buffer, buttons with the same label are considered separate views of the same button and so behave exactly alike. Since button labels are simply text strings, they may be embedded within any text to provide non-linear information or operational access points.

A button is selected by pressing a *Smart Key* within its label. The maximum length of a button label is limited by the variable *ebut:max-len*.

`button selection`
: The act of designating a Hyperbole button upon which to operate.

`category` A high-level, conceptual grouping of Hyperbole buttons into classes. *Implicit* and *explicit* groupings represent categories.

`class` A group of functions and variables with the same prefix in their names, used to provide an interface to an internal or external Hyperbole abstraction.

`context` A programmatic or positional state recognized by Hyperbole. We speak of Smart Key and implicit button contexts. Both are typically defined in terms of surrounding patterns within a buffer, but may be defined by arbitrary Emacs Lisp predicates. (Context may come to have a broader meaning within future versions of Hyperbole.)

See *Hyperbole environment*.

`explicit button`
: A button created and managed by Hyperbole. By default, explicit buttons are delimited like this `<(fake button)>`. Direct selection is used to operate upon an explicit button.

`global button`
: A form of explicit button which is typically accessed by name rather than direct selection. Global buttons are useful when one wants quick access to actions such as jumping

to common file locations or for performing sequences of operations. One need not locate them since they are always available by name, with full completion offered. All global buttons are stored in the file given by the variable *gbut:file* and may be activated as regular explicit buttons by visiting this file. By default, this is the same as the user's personal button file.

`global button file`
: See *button file, personal.*

`hook variable`
: A variable that permits customization of an existing function's operation without the need to edit the function's code. See also the documentation for the function `(run-hooks)`.

`Hyperbole`
: A flexible, programmable information management and viewing system built on top of GNU Emacs. It utilizes a button-action model and supports hypertextual linkages. Hyperbole is all things to all people.

`Hyperbole environment`
: A programmatic context within which Hyperbole operates. This includes the set of Hyperbole types defined and the set of Hyperbole code modules loaded. It does not include the set of accessible buttons. Although the entire Emacs environment is available to Hyperbole, we do not speak of this as part of the Hyperbole environment.

`hypertext`
: A text or group of texts which may be explored in a non-linear fashion through associative linkages embedded throughout the text. Instead of simply referring to other pieces of work, hypertext references when followed actually take you to the works themselves.

`implicit button`
: A button recognized contextually by Hyperbole. Such buttons contain no button data. See also *implicit button type.*

`implicit button type`
: A specification of how to recognize and activate implicit buttons of a specific kind. Implicit button types often utilize structure internal to documents created and managed without Hyperbole assistance, for example, programming documentation. *Ibtype* is a synonym for implicit button type. See also *system encapsulation.*

`instance number`
: A colon prefaced number appended to the label of a newly created button when the button's label duplicates the label of an existing button in the current buffer. This number makes the label unique and so allows any number of buttons with the same base label within a single buffer.

`link` A reference from a Hyperbole button to an entity. The referenced entity is sometimes called a *node* or *referent*. A specific class of actions which display entities are called *links*, such as a link to a file.

`local button file`
: See *button file, local.*

`mouse button`

`mouse key` See *Smart Key.*

`node` See *link.*

`predicate`
: A boolean (nil = false, non-nil = true) Lisp expression typically evaluated as part of a conditional expression.

`referent`
: See *link*.

`Smart Key`
: A context-sensitive key used within Hyperbole and beyond. Actually, there are two Smart Keys, a primary and a secondary Smart Key, sometimes referred to as the Smart Key and Smart Meta Key, respectively. The primary Smart Key, typically bound to the middle mouse key, activates Hyperbole buttons and scrolls the buffer forward a screenful when pressed at the end of a line. The secondary Smart Key, typically bound to the right mouse key, explains what a Hyperbole button does or scrolls the buffer backward a screenful when pressed at the end of a line.

  To see what a Smart Key will do within a particular context, depress and hold the key at the point desired and depress the other Smart Key. A buffer containing a description of its contextual function will then be displayed. You may release the two keys in any order after you have them both depressed. A press of the secondary Smart Key in an unsupported context displays a summary of Smart Key functions in each context, as does the Doc/SmartKy menu item.

`source buffer / file`
: The buffer or file within which a Hyperbole button is embedded.

`system encapsulation`
: Use of Hyperbole to provide an improved or simply consistent user interface to another system. Typically, implicit button types are defined to recognize and activate button-type constructs managed by the other system.

`view`
: A perspective on some information. A view can affect the extent of the information displayed, its format, modes used to operate on it, its display location and so forth.

`view spec`
: A terse (and to the uninitiated, cryptic) string that specifies a particular view of a link referent or some other information.

6 Personalized Information Environments (PIEs)

6.1 Overview

> "...it is useful to talk about providing an individual with a private intellectual work space...the intellectual workshop in which one does his collaborative bit within his working environment: one needs work spaces, tools to suit a myriad of tasks, places to store working materials, aids to hold them for examination and shaping–and they all should be easy to reach, quick to adjust to the task, easy to keep track of, etc. Interactive computer aids will have very significant effects here." [*Engelbart70*]

PIEmail and Hyperbole have been designed with a larger picture in mind, that of an overall Personalized Information Environment. PIEs are still very much an evolving concept, so no design yet exists. This whole chapter may be thought of as the future research directions section so common in papers today. We use it to describe a basic PIE model together with a standard set of components that we believe PIEs should have. As more of a PIE is developed, parts of PIEmail and Hyperbole would be separated out for more general use by the rest of the environment. This is one reason for doing object-oriented designs, to permit a loose coupling of components that aids in reimplementing them or shifting their place in an architecture.

PIEs are built from an assemblage of Tools, which are relatively independent, targeted applications, and from Managers, which provide inter-tool environment support services. Managers and Tools need not reflect large behavioral differences; in fact, we refer to both as PIE programs. The main differences are that Managers should rely on other Managers but not on Tools. Tools may also rely on Managers but only rarely on other Tools. Managers are typically an expected part of a PIE whereas Tools are added in or removed as needed. When we say that a Tool relies on a Manager, we mean that the Tool cannot operate without the Manager's facilities. Tools often will communicate with other Tools, such as when a Mail Tool messages an Image Tool to display a graphic sent as a piece of mail. The Mail Tool should still behave reasonably if the Image Tool is unavailable, however.

Our PIE model stems from the tool-based model in which programs are dedicated to specific tasks so that they may handle them well and may be built upon to exhibit more complex behaviors. This parallels the object-oriented model in which objects present abstract interfaces that may be generalized, specialized and related to one another in order to build large systems. Shared behaviors stimulate consistency throughout a working environment and reduce custom tool development time. Users gain the consistency they desire without sacrificing the reusability that programmers need to be productive. This perspective also yields models that closely parallel the activities and entities being modeled. Program users who are familiar with the modeled domain can utilize their prior knowledge to navigate the system, since they start with a conception of what recognized objects should do and how they interrelate to other parts of the system.

The point of developing PIEs is to simplify access to complex information spaces while also providing richer means of managing and retrieving distributed information components. Even when a user is saddled with isolated tools which work only with individual media types, a PIE framework should allow him to create aggregate multimedia structures, each of whose components maintain links to the tools which can perform operations on them [*HP89, CaSh91*]. Then the user should be able to think in terms of and work with these aggregate cross-media components.

6.2 Why Start with E-mail?

Electronic mail is an excellent medium upon which to build initial PIE support. It has emerged in research communitites as a primary form of both local and wide-area communication. Many business communities are also becoming heavy electronic mail users. So a ready-made market of electronic mail (e-mail) users exists, is familiar with basic techniques and inter-personal protocols for using e-mail, and needs access to information sources on a daily basis.

E-mail is a unique medium in that it works well for small working groups and large, distributed interest groups. The process of generating and sending a message is nearly the same whether it is intended for a single person or a thousand. Messages are delivered and stored with no human intervention. Recipients are free to read the messages whenever they want, as frequently as they want. Information sent can be further edited and included in other messages, unlike facsimile or video broadcast transmissions or even voice mail.

But virtually all electronic mail systems are built around the short-term processing needs that people have, with a view of information as a disposable commodity. It is true that many messages are short, concern a single topic, and are useful only during a particular time period. But this is only part of the picture. PIEmail systems help extend e-mail to support longer-term and greater capacity information access needs.

Our many years of participation in e-mail communities lead us to believe that limitations in the tool sets for handling mail constrain the types of information sent and tasks performed via e-mail. We expect to improve communication clarity and to simplify collaborative needs in local and remote groups by extending the support offered to e-mail users,

PIEmail systems include extensive automated mail handling facilities to shift messaging control from the hands of the senders to the receivers, who rightly should decide how they let such communications affect their work prioritization. Only the receivers have all the information needed to make such decisions.

Although this idea may seem basic to some, many traditional communication vehicles are not designed this way. For decades, telephone systems have required both immediate response and interaction with an unknown caller; innovations beyond simple, bulk message-taking are still slow in coming. Broadcast radios, such as the ones police use, are also similar. Registered mail and package deliveries require immediate attention in the form of a signature or the items are not delivered. In most of these cases, the sender/caller pays the service provider, which provides more than enough incentive to support his needs over the receiver's. Personalization is thus discouraged because the receiver does not support the market. Here again we see the uniqueness of electronic mail because we can support personalization at a variety of levels, regardless of who supports the delivery infrastructure (typically both sender and receiver support e-mail infrastructure). Delivery is already automatic and once a message is delivered, the recipient can fully dictate any set of workable rules to handle it.

6.3 Personalization Needs

We should explore what we mean by personalization, as this is the key term that separates our work from that of many others. Personalization is the adaptation of an entity to meet individual or local needs. Personalized information applications are task-specific information management or access tools designed to be tailored for such needs. Personalized information environments are information system service frameworks that can be shaped both manually and automatically to meet personalized needs.

A personalized system has to mold to its user's wishes. For information management systems, that means letting the user get to information in a way he finds convenient and helping him tie together informational chunks from multiple sources so that he can later access them as a single, coherent chunk. But what kind of needs are we talking about? We categorize PIE needs along several lines: user skills, organizational conventions, and growth paths.

6.3.1 User Skills

Any successful work in integrated environments must begin from the premise that users bring their own skill bases to any work they do. The tradition of documenting and training people to use new computer applications as if they know nothing about how to interact with a computer and its tools is no longer sufficient. If we do so, we not only waste the skills that people have but we engender a high-level of cognitive dissonance that manifests itself as resistance to the improvements offered by an environment. This is the problem seen by DVORAK keyboard activists in a QUERTY-educated world.

PIEs approach this problem on several fronts.

- PIEs permit incremental integration into a work space.

  Rather than require people to convert wholesale to a frightening new environment, PIEs permit a high level of experimentation and use of individual tools before one need choose to adopt others. Thus, a PIE mail tool can be used independently of a PIE calendar or database retrieval tool. But the shared service nature of PIEs encourages users to expand their PIE usage as they learn that skills developed through use of one tool guide them in their use of others.

- PIEs encourage personalized user interface adaptation.

  The first task in adding a new tool to an information work space is to map user interface components to tool operations. Many applications today allow customization such as key bindings, which link key presses to application operations. PIEs allow an additional higher-level, semantic binding which link event types to operation types. A change in such a binding affects all of the PIE programs that utilize it, obviating the need to individually change bindings which have the same semantics. System-wide object-oriented messaging which provides dynamic operation lookup makes this facility straightforward to implement.

- PIEs support differing skill bases and levels through multi-modal interfaces.

  An event driven architecture can allow different devices to generate the same events. So one could allow selection of an information artifact via keyboard, mouse or programming. Each device need only generate the event, so supporting multiple devices or user interface styles requires little extra code.

6.3.2 Organizational Conventions

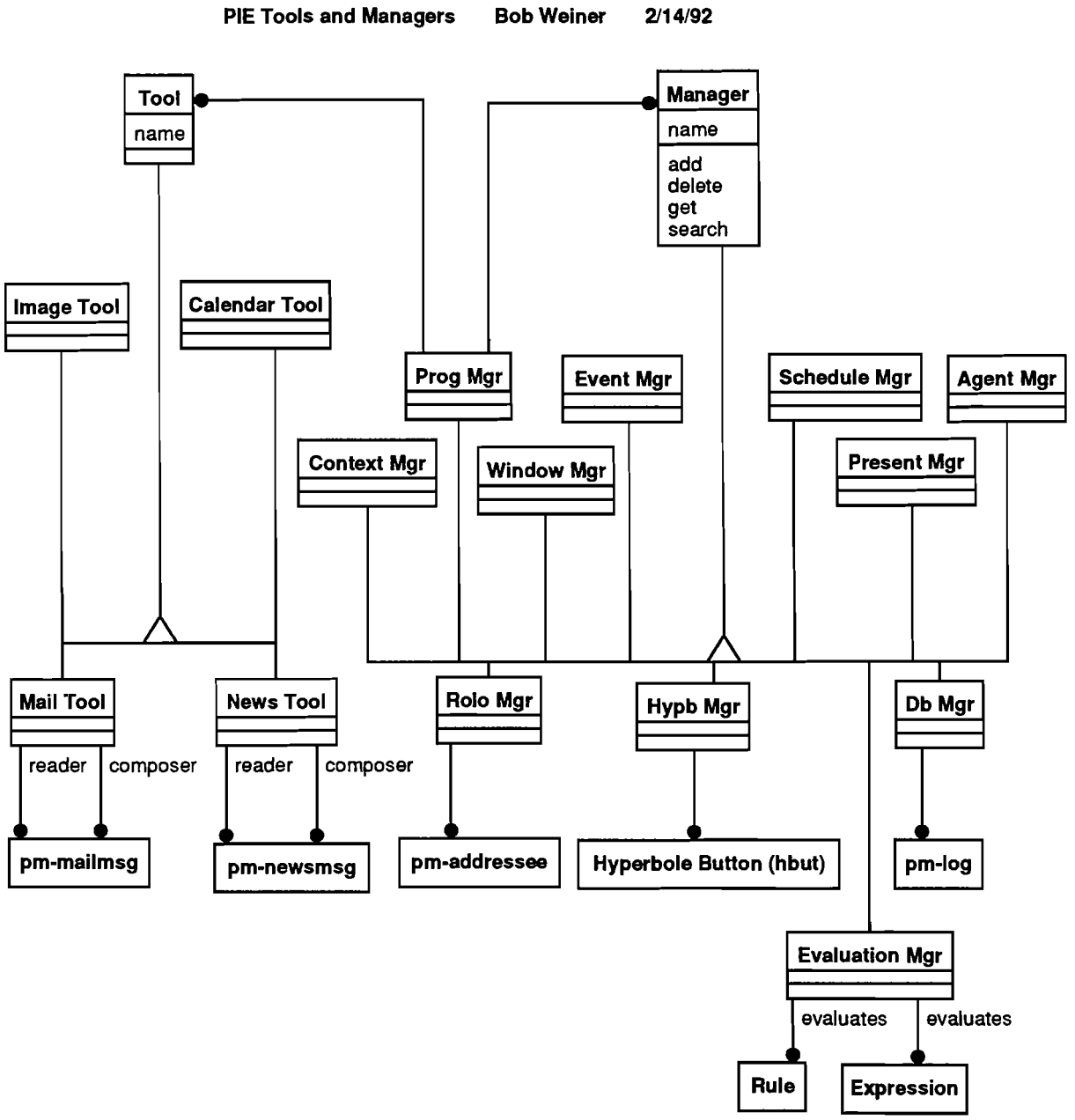
We also use personalization to refer to adaptation to local group needs. For example, maintenance of an electronic meeting announcement form and database would likely be administered by a group. This emphasis may seem at odds with our ideas for individual personalization, but we instead see it as another level of personalization. An individual remains free to further personalize his views and access tools for such meeting announcements, but management of the information remains at the group level where it belongs. Personalization is thus a relative concept that must always be examined in the context of how broadly information is to be shared.

6.3.3 Growth Paths

Productive workers continually expand their knowledge and skill bases. PIEs enhance their ability to do so. But one must still be concerned about developing past the point where a PIE can help. If such a point can be reached quickly by particular groups of people, they will have little incentive to migrate towards PIE usage.

PIEs are therefore open-ended, to deal with the variety of initial skill bases and developmental rates. Since one's knowledge base continues to expand, PIEs should not limit the knowledge bases that one can acquire or tap. User interfaces and media supported by PIEs should also be expandable to support new capabilities that emerge in these areas.

6.4 Managers and Tools



The above diagram illustrates a sampling of Managers and Tools that we envision composing a PIE. Let us take a look at the purposes of each Manager and Tool shown. Be aware that all of the relations among the classes in this diagram are not shown, as it is meant only to illustrate the PIE idea, not to provide a design representation.

6.4.4 Managers

PIE Managers provide core PIE functionality since their facilities are available to any existing or newly designed PIE Tools. In this respect, they serve as high-level reusable components; most classes being lower-level components of reuse.

The few methods shown for the Manager class in the previous diagram serve to add or remove Managers from a PIE, to return a handle to a Manager for communication purposes, and to search over the entities that a Manager controls according to some matching pattern.

6.4.4.1 Agent Manager

When one considers the sheer volume of information being made available electronically as international networks continue their phenomenal growth, it is clear that PIEs must include an agent-oriented facility that permits the specification of and asynchronous deployment of agents which pursue information management and retrieval goals.

An Agent Manager keeps track of agent availability, activation, and distribution. Availability features allow other PIE pieces to make queries as to locally or remotely available agents. They can then request activation of an agent with any necessary distribution parameters, such as a network search space over which to limit searches, e.g. the brown.edu domain. By the same token, the Manager should be able to recall agents by messaging them to cease in their active duties.

6.4.4.2 Context Manager

Contexts provide a persistent facility for maintenance of one's working artifacts. Environmental contexts allow one to easily put away a work in progress that utilizes a number of tools, and then easily to restore it when work is to continue. Much like the Rooms/3D Rooms work described in [*HeCa86, CaRoMa91*], but without any specific emphasis on graphical display of contexts.

A Context Manager should allow one to take snapshots of a PIE's state and then later restore these contexts. A simple scenario is to save a PIE context at system logout time and to restore it upon login. A more useful working scenario is to establish different contexts for different work tasks and to jump among them at will. The Window Manager should work with the Context Manager to permit the display of only those programs and entities needed within each context.

PIE Tools should utilize services offered by the Context Manager to permit straighforward saving and restoring of their contexts. Then one could restore Tool contexts either individually or in groups.

PIE contexts like all PIE entities may be named and therefore retrieved by name. The particular interface details are left to fit within a specific PIE design.

6.4.4.3 Database Manager

The Database Manager should offer an object-oriented database framework so that it may be useful in PIE entity and program management. It should in fact be used as a bootstrapping facility in the construction of a PIE as a means of maintaining object integrity and providing query facilities with which to inspect environmental facilities.

6.4.4.4 Evaluation Manager

The Evaluation Manager works closely with the Event Manager. The two may, in practice, be merged. It evaluates expressions, which the Event Manager or other programs may generate, and returns the result to its input source. These expressions are often used to affect the PIE state.

This Manager also handles environmental rules. Rules are just a special form of expression that often are chained together so that one rule triggers evaluation of others.

6.4.4.5 Event Manager

The Event Manager registers events that Tools and Managers are interested in receiving and then messages each as events of all types come into the environment. It should support some form of event prioritization and may utilize different dispatch strategies at different times, e.g. directing of all events of one type to a single Tool.

6.4.4.6 Hypertext Manager

This should offer core hyperlinking services that are useful to all different types of mediums that PIEs may support. Hyperbole is a good example of such a manager. By connecting it to editors that support other media types, one could support hyperlinks in their entities that would be managed by Hyperbole.

6.4.4.7 Presentation Manager

This, like the Agent and Database Managers, is a major project unto itself. The Presentation Manager permits specification of personal presentation requirements and then works with Tools to match output requests against these requirements. Filtering, ordering, routing, and formatting information would all be part of this Manager. PIEmail demonstrates a number of presentation-level personalization features with its view capabilities.

6.4.4.8 Program Manager

The Program Manager manages the start up and termination of PIE programs. PIE inter-program messaging allows event-based activation of programs. So, for example, activating a hyperlink may invoke a program that is needed to view its referent. Were the program to already be active, however, it would simply be messaged to display the referent.

6.4.4.9 Rolodex Manager

This is a much simpler program than the Database Manager. It is meant to provide reasonable access to categorized lists of items, like name and address records, without the overhead or complexity of a typical database system. Its simplicity facilitates its use in a wider variety of Tools and Managers. The rolodex used within PIEmail and Hyperbole is an example.

6.4.4.10 Schedule Manager

The Schedule Manager is used to register time-based actions. It is similar to the Event Manager and the two may be combined in some PIEs. An action is registered to occur either at a particular time or on a recurring basis. The action consists of one or more messages routed through the Event Manager to one or more Tools that have registered to respond to the messages or to which it is directly addressed.

There should be a means of deferring actions if needed Tools are unavailable when actions trigger. For example, if a Mail Tool is a client resident on a non-networked personal computer that periodically connects to a mail server where a Schedule Manager runs, one might want to update the client's address book from the server's through a scheduled action. But this would not be possible until the client connected to the server and so could not in general be scheduled.

6.4.4.11 Window Manager

The Window Manager provides PIE programs with control over window placement, sizing and other window attributes. It should at least support both tiled and overlapping window styles. It should permit grouping of windows and allow operations to be applied to these groups.

6.4.5 Tools

We describe only a few sample Tools below as examples of what a PIE might offer.

6.4.5.1 Calendar Tool

A Calendar Tool would manage a person's work schedule. A more useful tool would also provide group and resource scheduling facilities. Appointment reminders could make use of the Schedule Manager's services. Calendar entries could be managed with the Database or Rolodex Manager whereas calendar views could utilize Presentation Manager mechanisms.

6.4.5.2 Image Tool

An Image Tool should display images and convert them among several formats for display. It should permit editing component-based images and support hyperlinks to and from image components.

6.4.5.3 Mail Tool

A Mail Tool should offer personalized electronic mail handling along the lines of our PIEmail prototype. A full PIE will permit a much simpler implementation of such a tool because of all the environmental support.

6.4.5.4 News Tool

A News Tool should support category-based browsing and submission of news articles. Its operation would be very similar to PIEmail, which can read news articles but does not offer a fully general news reader. A good design could merge the operation of the Mail and News Tools and simply dispatch appropriately whenever a mail folder or newsgroup is opened and browsed.

7 Prior Work

There is way too large a body of work on electronic mail and information mangement systems for us to examine or even to cite. Instead we look at a cross-section of prior efforts that relate well to our work and leave the reader to explore publications on these projects if a broader view and more references in these areas are desired.

We first examine Doug Engelbart's augmentation research because it relates well to our overall PIE ideas. Then we go on to more specific work germaine to PIEmail systems: the Andrew Message System, Information Lens, Walnut, and Tapestry. Lastly, we point the reader to a few references that support some of our design decisions.

7.1 Engelbart and Augmentation

Doug Engelbart's multi-decade research has demonstrated the effectiveness of integrated information environments, hypertext links, multimedia mail, and personal workstyle support [*Engelbart84a, Engelbart84b, EnEn68*].

Engelbart was initially motivated to consider how to deal with the burgeoning flow and management of information that society was producing. He settled on the term augmentation to describe his work in seeking out both processes and environments that could enhance man's ability to perform knowledge work far beyond his capabilities when not augmented. This work was performed with a research team that he headed at Stanford Research Institute (SRI). It established the world of interactive computing and led to the development of the NLS system. The Augmentation Research Center (ARC), as it came to be called, did years of experiments with a number of cross-disciplinary teams in order to develop effective methods and tools. One should keep in mind the fact that NLS' architecture was based on the results of examining real-world issues, as we discuss it.

NLS had at its base a file store that emphasized hierarchically structured, node-based files so that one could efficiently organize and hyperlink bits of information. Each node (actually called a statement in NLS) had a permanent reference number attached that permitted its reference within a link. Section titles, paragraphs, and diagrams each would typically be treated as a node. Finer grained links could also be made to points within nodes. Links were embedded within text and were easily human readable.

The idea of gaining different views over an information space appeared early in this research. One would apply a view specification (viewspec) to a group of nodes to gain a particular perspective. For example, a table of contents view would show only section titles in a hierarchy. Links could include viewspecs to indicate how their referents should be displayed. A concise (cryptic to the uninitiated) notation was devised to allow quick specification and alteration of views.

Views and NLS's hierarchical structure made outline processing natural, permitting one to collapse and expand nodes and to operate on them in groups, as when one wants to move two sections from one chapter to another.

A window system was produced to support multiple views and inter-file comparison and linking on screen. The mouse was invented and joined with a 5-key chord keyboard, in addition to the regular keyboard, to provide rapid direct selection and command invocation using a two-handed interface. A verb-noun (action-target) command grammar, e.g. delete word, was developed and applied consistently throughout the core NLS environment and the many subsystems that grew around it.

The user interface was multi-modal and offered deep personalization features. Nodes could be operated upon via direct selection, by relative or absolute addressing, and by user-created macros. All events such as key or mouse button presses were mapped to commands through customizable grammar files specific to each NLS subsystem. If one tailored the command grammars to one's own needs, then all of the applications that utilized the grammar were immediately tailored also. This is similar to our notion of a semantic operation mentioned in the previous chapter but differs from the application-specific key bindings which are more common today.

Mail messages were treated as structured objects and could include links across the fledgling Internet, just like files. Documents could also be submitted via mail to a permanent library-type archive called the Journal as a form of electronic publication.

Some of these facilities are standard parts of today's computing environments, but NLS was the first implementation of many of the concepts. It is a much more extensive environment than we can portray here and remains today the only system we can point to that offers a firm architectural basis for a PIE. Unfortunately, NLS was never ported to modern hardware. As a result, only one remaining machine in the world runs NLS; it is used by Engelbart and his staff.

Hyperbole's integration with PIEmail offers many of the kinds of NLS features surveyed above, in slightly different forms. View specification work for Hyperbole is in progress and should offer an extensible, view mechanism with the same kinds of viewspecs as NLS. The major component of NLS that is unavailable with Hyperbole is the node-based filesystem. This limits to some extent the kind of in-file linking that can be done but does not hinder the kind of outline processing facilities that Hyperbole can utilize.

As a final note, we claim that Engelbart's research on augmentation systems offers much evidence for the value of PIEs. His work has shown that continuous feedback from user experience, to process organization, to system design, can produce distinctly non-linear gains in productivity [*Engelbart82*]. Personalization support at an environment level, rather than a per-application level, affords people a choice of working style and allows them to make effective use of it across many of the tasks they must perform. As the gains become noticeable, the incentive to work on feedback-based improvements increase until a point of diminishing returns, yielding a productivity enhancement process that can rapidly evolve towards desired goals. The power of this approach can be seen in [*Stone91*] which surveys a number of AUGMENT applications that show evidence of being far ahead of other work of their time. (AUGMENT was the commercial name for the NLS research environment.)

7.2 Andrew Message System

The Andrew Message System (AMS) [*RoEvBo87, BoTh88*] was built as an exemplar framework for the large-scale Andrew project at CMU. It offers a rich, message handling environment designed to encourage asynchronous collaboration among large, diverse sets of users. AMS was built from the Andrew Toolkit and so contains a number of hypermedia facilities, [*ShHaMcNe90*].

[*BoTh91*] describes the most commonly used mail reader built from AMS, Messages, focusing on its collaborative facilities. It supports folders and multimedia mail, including formatted text. Messages may be processed in groups just like PIEmail's sets. The accessability of message bases may be made private, public or limited to a specific group. Message bases are dealt with in a bulletin board fashion where people can create topic-based groups and then post to them, allowing readers to inspect them at their leisure. Presently, PIEmail has no specific support for shared message bases, but as mentioned in the chapter on PIEs, we foresee a later merging of mail and news handling under such as framework.

AMS active messages contain a programmatic component that can direct an interaction with a user through dialog boxes and other means. A sample interaction could have a user fill out a multiple choice survey and then mail it back to the sender.

FLAMES (Filtering Language for the Andrew MEssage System) is a Lisp-like language used within AMS to automate mail handling tasks, just like our rules concept but with a richer set of standard operators geared towards mail handling.

Borenstein and Thyberg, the authors of [*BoTh91*], go to great lengths to show that user interfaces can be made that deal well with both novices and experts, along the lines of our multi-modal interface requirements. They argue that ease of use can be treated independently from program complexity, that in fact, more complex programs can be easier to use than simpler ones. Our experience and prototyping work over the last several years has agreed with this axiom.

The discussion of the Advisor system built upon AMS highlights the importance of automatic message classification. Advisor is an electronic question and answer management system that allows users to mail any question to a single address. By classifying messages according to contents, the system internally directs the message as best it can and thereby can tap distributed expertise. Answers come back to the user from the single advisor address, again hiding what amounts to an internal optimization from the user base which has no need or desire to know about it. AMS also supports task-based addresses. In the Advisor system, users who are more knowledgable and conscientious about directing their questions can send them to specific advisor subtopics by using task addresses like, advisor+quota. (AMS uses a '+' rather than a '.' to separate the task part of an address.)

AMS and Messages represent years of work and testing by a number of CMU researchers. They demonstrate many of the same kinds of features found in PIEmail indicating a synergy in our view on message handling needs. They represent a more sophisticated environment than PIEmail overall and their goals are broader. But they are large and complex. Although AMS is included in the MIT X Consortium's distribution of the X Window System, few sites that have this distribution have in fact set it up for use because of this complexity and the management necessary.

PIEmail in contrast builds on the PIE tool philosophy of being able to incrementally integrate with an existing environment. It is fairly small for such a flexible tool and requires only a few seconds to load and to initialize. Individuals can choose to use it with very little administrator support. Therefore, we see it as a more accessible mail tool through which one can build as much sophistication as necessary to meet local needs.

7.3 Information Lens

Tom Malone's Information Lens work popularized the term, semistructured message, illustrating automatic processing of task-oriented message forms by using special purpose field information encoded in the message headers [*MaGrLa87*].

Information Lens provided personalized mail handling forms and rule-based processing on dedicated Lisp machines. Studies of Information Lens users demonstrated that rule-based e-mail handling can be useful to both light and heavy e-mail users [*MaMaCr89*].

Current work on Object Lens has a broader goal of allowing process modeling and automation by non-programmers, for example, sequenced message routing to automate a purchasing approval process. It is a prototype in Lisp on a Macintosh [*LaMa91*].

7.4 PARC Work

Xerox Corporation's Palo Alto Research Centers (PARC) has long been a leader in collaborative technology research. Many research efforts have targeted electronic mail handling as a key component of future office information environments. The Walnut mail manager [*DoOr85*] developed at PARC demonstrated the possibilities in using a full database system for mail management, most notably the ability to share messages and to query over message bases.

[*Terry90*] lays out high-level requirements for the ongoing Tapestry message handling environment being developed at PARC by discussing issues that researchers there have identified as important in the evolution of e-mail systems. In particular, he identifies seven requirements.

1. Give recipients control over what they read.
2. Free the sender from choosing recipients.
3. Provide shared storage.
4. Never delete anything.
5. Allow content-based retrieval.
6. Enforce sender-specified access controls on messages.
7. Convert messages into recipient-desired format.

These are quite different from our requirements, see Section 3.4.2 [PIEmail Requirements], page 18, and therefore bear future examination. PIEmail offers support for requirements 1, 5, and 7. The rest are really issues that if dealt with, must be handled within an environment framework rather than a single tool. They involve issues of administration, user naming services, and communication and file system security.

7.5 Miscellaneous

Although we expect PIEmail to be most appealing to heavy e-mail users, there is prior work that indicates a number of PIEmail's design features will work well for many audiences.

[*TsCh83*] discusses the value of filtering over message bases and more generally in taking an information retrieval perspective to message management.

[*JeRo87*] found that a forms-based interface when used to perform mail filtering could bring non-programmers up to the level of programmers who had to write filtering instructions procedurally. They further found that the programmers were faster than the non-programmers when using the forms-based interface. This coupled with success seen in Information Lens use suggests that forms-based queries and rule generation will work well for a diversity of users.

[*WiWhGoJo83*] report on an experiment that suggests that tailoring an e-mail reader user interface through repeated user feedback yields a much more effective interface. This idea is the reason for our strong emphasis on personalization.

The Consul/CUE [*KaMaSo83*] system points to the effectiveness of multi-modal interfaces, where users have a choice of interaction mechanisms for each possible command. Some argue that such user interfaces are confusing since users have trouble deciding which techniques to choose. Our anectdotal experience observing people using our systems over the last year suggests the opposite, however. People either have prior knowledge about what interaction techniques they prefer or they quickly settle on those with which they can be comfortable.

Multi-modal interfaces afford usability to a broader class of users. If one's interaction style changes, e.g. a mouse becomes unusable due to limited arm movement, one can switch to one of

the other available input devices. Personalized environments should provide multi-modal interfaces by separating commands from the interaction means used to execute them. Users should then be free to associate commands with the input techniques they are used to or find they prefer.

Conclusion

PIEmail offers an extensive set of contributions beyond traditional e-mail readers. We can summarize them as follows.

CONTRIBUTIONS	SUPPORT FACILITIES
+ Automated mail processing:	rules, task addresses
+ Executable mail components:	buttons/links
+ Grouping of messages:	folders, sets, priorities
+ Selecting message groups:	filters
+ Deep customization facilities:	rules, views, filters, sorters
+ Effective retrieval of mail:	queries
+ Conversation handling:	conversations
+ Addressee management:	rolodex

Hyperbole presents a working prototype of a powerful personal information manager. It offers extremely intuitive browsing capabilities and excellent extensibility to meet specific application demands. Its integration with PIEmail permits wide-area hyperlinking and saves on communication costs by lowering the number of necessary enclosures.

Our PIEmail design philosophy has been heavily user-centered for two reasons. We seek to demonstrate the important productivity impact that effective interfaces can have in collaborative, information-intensive tasks. We further want to make the systems appealing enough to encourage adoption as both end-user tools and frameworks for targeting specific mail and information delivery applications.

There is much work to be done to achieve the vision of an effective Personalized Information Environment. But our initial work on PIEmail and Hyperbole point out that strides can be made quickly and that even prototypical tools can effectively elevate user performance. Our broader expectation is that as personalization becomes a standard working component of peoples' tool chests, that we will see an entirely new generation of problems approached. As the complexity of these new problems becomes mentally manageable, we can leverage off the facilitation that our tools and working processes provide to tackle further issues, as Doug Engelbart has so long argued in his augmentation papers [*EnEn68, Engelbart70*].

Acknowledgments

Motorola Inc. provided us with the opportunity and financial support to pursue this work.

We had the pleasure of Peter Wegner's advising and Stu Hamlyn's inputs throughout the course of this project. We owe a great debt to Mark Lambert of Oracle Corp. and formerly of MIT who provided his work on the pcmail package that formed a base for our PIEmail prototype. Richard Stallman's and untold others' work on the Free Software Foundation's GNU Emacs environment also made the prototyping a rewarding process.

Doug Engelbart of the Bootstrap Institute patiently explained his work and thereby gave us a larger, more cohesive vision within which to see our own. Nathaniel Borenstein of Bellcore and formerly of CMU kindly provided us with background information on the Andrew Message System and MIME. Jim Rumbaugh and GE made OMTool available to our department, with which we produced our design diagrams. The Human-Computer Interaction Bibliography Project organized by Gary Perlman of the Ohio State University pointed us to a number of references [*Perlman91*].

Finally, the Brown Computer Science Department supplied a wonderful intellectual and work environment throughout our degree program.

References

[BoFr92] N. S. Borenstein and N. Freed. *MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies-Mechanisms for Specifying and Describing the Format of Internet Message Bodies.* Published as Internet Drafts, June/October/December/February, 1991/92, Proposed Internet Standard (expected April 1992).

[BoTh88] N. S. Borenstein and C. A. Thyberg. Cooperative Work in the Andrew Message System, *Proceedings of the CSCW'88 ACM Conference on Computer-Supported Cooperative Work*, 1988, pp. 306-323.

[BoTh91] N. S. Borenstein and C. A. Thyberg. Power, Ease of Use and Cooperative Work in a Practical Multimedia Message, *International Journal of Man-Machine Studies*, 1991, 34(2), pp. 229-259.

[CaRoMa91] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The Information Visualizer, An Information Workspace, *Proceedings of the ACM CHI'91 Conference on Human Factors in Computing Systems*, 1991, pp. 181-188.

[CaSh91] R. Carr and D. Shafer. *The Power of PenPoint.* Addison-Wesley, Reading, MA, 1991.

[Crocker82] D. H. Crocker. *Standard for the Format of ARPA Internet Text Messages.* Internet Request for Comments Number 822 (RFC-822), Department of Electrical Engineering, University of Delaware, August 13, 1982. Available from /anonymous@ftp.nisc.sri.com:rfc/rfc822.txt.

[DoOr85] J. Donahue and W. Orr. *Walnut: Storing Electronic Mail in a Database.* Xerox Palo Alto Research Centers Technical Report, CSL-85-9, November 1985.

[EnEn68] D. C. Engelbart and W. English. A research center for augmenting human intellect, *Proceedings of the Fall Joint Computer Conference*, 33, 1, AFIPS Press: Montvale, NJ, 1968, pp. 395-410.

[Engelbart70] D. C. Engelbart. Intellectual Implications of Multi-Access Ccmputer Networks, *Proceedings of the Interdisciplinary Conference on Multi-Access Computer Networks*, Austin, Tx., April, 1970. (AUGMENT,5255,)

[Engelbart82] D. C. Engelbart. Toward High-Performance Knowledge Workers, *Proceedings of the AFIPS Office Automation Conference (OAC '82 Digest)*, San Francisco, CA, April 5-7, 1982, pp. 279-290. (AUGMENT, 81010,) Also published in: I. Greif, editor. *Computer Supported Cooperative Work: A Book of Readings.* Morgan Kaufmann Publishers, San Mateio, CA, 1988, pp. 67-78.

[Engelbart84a] D. C. Engelbart, Authorship Provisions in Augment. *Proceedings of the 1984 COMPCON Conference (COMPCON '84 Digest)*, February 27-March 1, 1984, San Francisco, CA. IEEE Computer Society Press, Spring, 1984, pp. 465-472. (OAD,2250,)

[Engelbart84b] D. C. Engelbart. Collaboration Support Provisions in Augment. *Proceedings of the AFIPS Office Automation Conference (OAC '84 Digest)*, Los Angeles, CA, February 1984, pp. 51-58. (OAD,2221,)

[Hamlyn92]
S. Hamlyn. *NeXTStep PIEmail Proposal.* Brown University, Providence, RI, 1992, unpublished.

[HeCa86] D. A. Henderson, Jr and S. K. Card. Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window-Based Graphical User Interface, *ACM Transactions on Graphics*, 5(3), 1986, pp. 211-243.

[Horton83]
M. Horton, *Standard for Interchange of USENET Messages.* Internet Request for Comments Number 850 (RFC-850), USENET Project, June, 1983. Available from /anonymous@ftp.nisc.sri.com:rfc/rfc850.txt.

[HP89] HP NewWave Issue, *Hewlett-Packard Journal*, August 1989.

[JeRo87] R. Jeffries and J. Rosenberg. Comparing a Form-Based and a Language-Based User Interface for Instructing a Mail Program, *Proceedings of the ACM CHI+GI'87 Conference on Human Factors in Computing Systems and Graphics Interface*, 1987, pp. 261-266.

[KaMaSo83]
T. Kaczmarek, W. Mark, and N. Sondheimer. The Consul/CUE Interface: An Integrated Interactive Environment, *Proceedings of the ACM CHI'83 Conference on Human Factors in Computing Systems*, 1983, pp. 98-102.

[KaLoCaLa92]
S. Kaplan, C. Love, A. Carroll, D. LaLiberte. *Epoch: GNU Emacs for the X Windowing System.* Release 4.0, University of Illinois, Urbana, IL, March 1992.

[LaMa91] K. Lai and T. W. Malone. Object Lens: Letting End-Users Create Cooperative Work Applications, *Proceedings of the CHI'91 ACM Conference on Human Factors in Computing Systems*, 1991, pp. 425-426.

[Mackay88]
W. E. Mackay. More than Just a Communication System: Diversity in the Use of Electronic Mail, *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, 1988, pp. 344-353.

[MaGrLa87]
T. W. Malone, K. R. Grant, K. Y. Lai, R. Rao, and D. R. Rosenblitt. Semistructured messages are surprisingly useful for computer-supported coordination, *ACM Transactions on Office Information Systems*, 1987, 5(2), pp. 115-131.

[MaMaCr89]
W. E. Mackay, T. W. Malone, K. Crowston, R. Rao, D. Rosenblitt, and S. K. Card. How Do Experienced Information Lens Users Use Rules?, *Proceedings of the ACM CHI'89 Conference on Human Factors in Computing Systems*, 1989, pp. 211-216.

[Perlman91]
G. Perlman. The HCI Bibliography Project, *ACM SIGCHI Bulletin*, 23(3), 1991, pp. 15-20.

[RoEvBo87]
J. Rosenberg, C. Everhart, and N. Borenstein. An Overview of the Andrew Message System. *SIGCOMM '87 Workshop*, Stowe, VT, August, 1987.

[RuBlPrEdLo91]
J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[ShHaMcNe90]
M. Sherman, W. J. Hansen, M. McInerny, T. Neuendorffer. Building Hypertext on a Multimedia Toolkit: An Overview of Andrew Toolkit Hypermedia Facilities, *Proceedings of the ECHT'90 European Conference on Hypertext*, 1990, pp. 13-24.

[Shneiderman82]
B. Shneiderman. The future of interactive systems and the emergence of direct manipulation, *Behavior and Information Technology*, 1982, 1(3), pp. 237-256.

[Stallman87]
R. Stallman. *GNU Emacs Manual.* Free Software Foundation, Cambridge, MA, March 1987.

[Stone91] D. Stone. *AUGMENT's Support of Organizations: A Brief History.* Bootstrap Institute, Palo Alto, CA, 1991. (AUGMENT,132615,)

[Terry90] D. B. Terry. *7 Steps to a Better Mail System.* Xerox Palo Alto Research Centers Technical Report, CSL-90-12, September 1990.

[TsCh83] D. Tsichritzis and S. Christodoulakis. Message Files, *ACM Transactions on Office Information Systems*, 1983, 1(1), pp. 88-98.

[Weiner92]
B. Weiner. *Hyperbole Manual.* Brown University, Providence, RI, 1992. Available electronically via anonymous ftp from: /anonymous@wilma.cs.brown.edu:pub/hyperbole.

[WiWhGoJo83]
D. Wixon, J. Whiteside, M. Good, and S. Jones. Building a User-Defined Interface, *Proceedings of the CHI'83 ACM Conference on Human Factors in Computing Systems*, 1983, pp. 24-27.

Appendix A Glossary

`Address` An electronic mail address. PIEmail supports Internet domain-based addressing standards. It also offers task-based, personally generated addresses.

`Attribute`
A value-based field attached to a PIEmail entity. Most commonly attributes are associated with individual messages. Many attributes are boolean-valued (true or false).

`Body` The content portion of a message. The header is the other portion. MIME-compliant message bodies may further be split into typed components.

`Button` A Hyperbole button. Buttons may be embedded within PIEmail messages, rolodex entries and within textual files. A button is activated by pressing a key while pointing at the button label. Buttons often link to local or wide-area information sources but may perform any rule-type action.

`Command` A user visible PIEmail operation.

`Conversation`
A series of messages that reference one another.

`Entity` PIE informational artifacts. PIEmail entities include: messages, rolodex entries, and folders.

`Filter` A preset mechanism for removing messages from a set. See also Query.

`Folder` A persistent message store, used to store messages by category.

`Form` A field-oriented screen specification for data entry or display.

`Header` The structured, envelope part of a message at its beginning.

`Hyperbole`
An associative, button-based information manager integrated with PIEmail.

`Mailbox` A store to which mail transfer agents deposit mail. PIEmail reads mail from mailboxes and stores it in folders which classify and add attributes to it.

`Mail User Agent (MUA)`
An interactive program used to read and send electronic mail messages. Neither MUA nor MTA should be confused with the more technical use of the term agent to mean an autonomous process that carries out a plan or strategy on behalf of a user.

`Mail Transfer Agent (MTA)`
A non-interactive program that transfers mail from one network node to another along a route.

`Message` An piece of electronic mail. It consists of a header and a body.

`MIME` Multipurpose Internet Mail Extensions. An evolving Internet standard for transfer and encoding of multi-media mail. PIEmail supports some MIME features with an eye towards broader MIME compliance in the future.

`PIE` Personalized Information Environment. An integrated, tool-based approach to maximizing electronic information handling productivity in a technical realm. PIEmail and Hyperbole are prototypical pieces of a PIE.

`Priority` An urgency level assigned to a PIEmail entity. Priorities may be given as names or numbers. Lower number represent higher priorities, with 1 being the highest.

`Query` A request to retrieve messages satisfying specified criteria.

`RFC-822` This specifies the base-level standard electronic mail format used on the Internet. See also MIME.

`Rolodex` An addressee and contact management facility integrated with PIEmail.

`Rule` A predicate-action pair used under PIEmail to process messages.

`Sendmail` The Berkeley UNIX mail transfer agent used by many Internet hosts. A simple customization to a Sendmail mail processing rule set permits the use of task addresses under PIEmail. See Appendix D [Sendmail Tasks], page 81.

`Set` A set of items. Most commonly a set of messages to which PIEmail operations are applied.

`Sorter` A named method for sorting entities.

`Subfolder`
A folder that is a specialization of its parent.

`Task` A work task that is assigned a PIEmail address to simplify rule processing and categorization.

`View` A PIEmail construct that provides a perspective on a set of entities.

Appendix B Class Diagram Notation

The diagrams used herein utilize the notation from [*RuBlPrEdLo91*]. Their notation is in turn derived from the entity-relationship notations that relational database modelers have long used, but refined for use with object-oriented models. In the text, Rumbaugh et al. discuss three different design views and notations: an object model, a dynamic model and a functional model. For our purposes in this work we have used only the object model, and only a subset of its notation that we explain here.

Boxes represent classes, although in the diagrams labeled with the term, concept, they represent abstract functional units. Sometimes class boxes are segmented into three parts, which from top to bottom are: the class name, its attributes, and its methods. In most cases we omitted the attributes and methods, to emphasize just the inter-class relationships. (In the Hyperbole design diagram, we included them in the button classes to show a bit more detail of how these classes work.)

Relations between classes are shown as lines connecting one class to another. The role that one class plays with regard to another may be included as a label near its relation end point. The arity of the relation is shown through the relation's end points. A black circle means zero or more; a white circle means zero or one; and no circle means one. So a line with no endpoints indicates a one-to-one relationship. Arrowheads are used just to indicate role directionality more clearly. Diamonds represent an aggregation in which the diamond is attached to an aggregate class whose objects are comprised in part from objects of the class at the other end of the relation.

Class inheritance is shown via a relation with a triangle in the middle of it. The triangle points toward the parent class and is connected to all of the subclasses.

Comments are included where thought to be helpful within the open space of the diagrams.

That is the entirety of the diagrammatic notation used. Anyone familiar with object-oriented concepts should have an easy time with it. [*RuBlPrEdLo91*] is a good reference for anyone who is not.

Appendix C Hyperbole Demonstration

This is the text of the interactive Hyperbole demonstration that comes with the Hyperbole distribution. We hope eventually to have a similar demonstration that covers PIEmail.

C.1 Overview

This file demonstrates simple usage of the basic Hyperbole button-action types and shows how Hyperbole can support a style of self-documenting, interactive files. See the glossary in The Hyperbole Manual, '`(hypb.info)Glossary`', if terms used here are unfamiliar to you.

C.2 Smart Key Handling

This button prints the <(factorial)> of 5 in the minibuffer when activated with the primary Smart Key, normally the middle mouse button. (Once you have Hyperbole installed, just click on the word, <(factorial)>.) If you use the secondary Smart Key, normally the right mouse button, you get help for the preceding button. The help provides a summary report of the button. You will see that it utilizes the 'eval-elisp' action type. You can also see who created it. Try it.

Note that the create-time and mod-time are displayed using your own timezone but they are stored as universal times. So if you work with people at other sites, you can mix their buttons with your own within the same document and see one unified view of the modification times on each button. These times will also be useful for sorting buttons by time when such features are provided in future Hyperbole releases.

To display the top-level Hyperbole menu, click the Smart Key anywhere within this paragraph or alternatively, use {C-h h}. {q} or {C-g} will quit from the menu without invoking any commands if you just want to take a look. A menu item is selected by pressing the Smart Key over it or by typing its first letter in upper or lower case.

A click of the secondary Smart Key on a menu item gives help on it. A click of the same key within this paragraph, when a Hyperbole menu is not displayed, pops up a summary of all the things the Smart Keys can do. Try it.

C.3 Explicit Button Samples

If your <(Info-directory-list)> or <(Info-directory)> variables include the directory that contains the online GNU Emacs manual, activation of the next button will tell you about <(keyboard macros)>. Can't remember a Hyperbole term? Check out the Hyperbole Manual <(glossary)>.

Here is a <(keyboard macro)> button that shows documentation for the first parenthesis delimited Emacs Lisp function that follows it, e.g. `(hbut:report)`. You can see that a button label can consist of a number of words, up to a set <(maximum length)>.

A <(shell command)> button can do many things, such as display the length of this file. While such commands are executing, you can perform other operations. If you create a button that runs a shell command which displays its own window, use `exec-window-cmd` rather than `exec-shell-cmd` as its action type.

You can link to files such as your <(.login)> file. Or directories, like the <(tmp directory)>. When creating file links, if the file you are linking to is loaded in a buffer, you are prompted as to

whether you want the link to jump to the present point in that buffer. If so, the link will always jump there, so position point within the referent file to take advantage of this feature. Note how a separate window is used when you activate file link buttons. Most basic Hyperbole action types display their results in this manner.

You can make a button an alias for another by using the `link-to-ebut` action type. This <(factorial alias)> button does whatever the earlier <(factorial)> button does.

The `link-to-mail` action type allows you to reference mail messages that you have stored away. We can't demonstrate it here since we don't have the mail messages that you do.

Hyperbole buttons may also be embedded within mail messages. Even buttons copied into mail replies can work:

> Emile said:
> >
> > Hyperbole is better than home baked bread but not as filling.
> > The following sample button displays a message, <(as long as you
> > click within its first line)>.

C.4 Implicit Button Samples

C.4.1 Implicit Path Links

Any doubly quoted pathname acts as an implicit button that displays the referenced path. These are `pathname` implicit buttons. For example, activate '`README`'.

If you use the Andrew File System or the ange-ftp add-on to GNU Emacs, such remote pathnames will work as well. (The latest version of ange-ftp may always be obtained via anonymous ftp to: '`/anonymous@alpha.gnu.ai.mit.edu:ange-ftp/ange-ftp.el.Z`'). Once you have loaded the ange-ftp package, if you are on the Internet, you can click on any of the following to browse the contents of the Hyperbole distribution at Brown University (limit the amount you do this so as not to deny others access to the archive):

```
"/anonymous@wilma.cs.brown.edu:pub/hyperbole/"
/anonymous@128.148.31.66:/pub/hyperbole/
# This one requires:  (setq ange-ftp-default-user "anonymous")
/wilma.cs.brown.edu:pub/hyperbole/
```

You can see that for ange-ftp pathnames, Hyperbole recognizes them with or without the double quote delimiters. These same pathnames can be used within explicit buttons which link to files or directories.

GNU Info (filename)node references such as '`(hypb.info)Glossary`' or '`(emacs)Glossary`', work similarly, thanks to the `Info-node` button type. Try one of the Glossary buttons above.

So now when browsing the many documents that refer to filenames or Info nodes in this way, you can just click on the name to see the contents. (If a doubly quoted string references a local pathname that does not exist within the file system, it will not be considered a pathname button by Hyperbole.) Pathname implicit buttons provide one example of how Hyperbole can improve your working environment without you having to do any work at all.

Hyperbole provides a history command which returns you to previous button locations in the reverse order of the way you traverse them. You access it by selecting the Hist command from the top-level Hyperbole menu, {`C-h h h`}.

Now suppose you want to browse through a number of files within the Hyperbole distribution. You could use the Emacs dired subsystem, '`(emacs)Dired`', but a faster way is to note that files named MANIFEST and DIR are used to summarize the files in a directory, so we can use each of their entries as an implicit button (of `dir-summary` type) to take us to the file.

Let's look at '`MANIFEST`'. Now click anywhere within a line in the '`MANIFEST`' file and you see that it is displayed as expected. You can get help on these buttons just like any others.

C.4.2 Bibliography Buttons

Here's a use of an bibliography reference implicit button which allows you to see a bibliography entry such as [*Stallman87*] when you activate the button between brackets.

C.4.3 Hyperbole Source Buttons

If you ask for help on the [*Stallman87*] button, the first line of the help buffer will look like this:

```
@loc> "DEMO"
```

except it will contain the full pathname of the file. If the button were embedded within a buffer without an attached file, the first line of the help buffer might look like:

```
@loc> #<buffer *scratch*>
```

If you click on the buffer name, the buffer will be displayed just as a file buffer would. This type of implicit button is called a `hyp-source` button.

C.4.4 UNIX Man Apropos Buttons

Below are some lines output by the UNIX '`apropos`' command (with a little touchup for display purposes). A button activation anywhere within such a line recognizes the line as an apropos entry and tries to display the man page for the entry. Try it. (If you happen to use the '`superman`' package which fetches man pages in the background, you'll have to wait for the next version of superman which removes incompatibilities with the standard man page fetch command before you can use these `man-apropos` implicit buttons.)

```
grep, egrep, fgrep (1V) - search a file for a string or regexp
rm, rmdir (1)           - remove (unlink) files or directories
touch (1V)              - update access and change times of a file
cat (1V)                - concatenate and display
```

C.4.5 Key Sequence Buttons

Any Emacs key sequence delimited with braces may be executed by activating it as a button, for example `{C-u C-p}` should leave point four lines above the button line. A help request upon the key sequence displays the documentation for its command binding, i.e. what it does.

If it does not represent a bound key sequence, it will not be treated as a key sequence button.

C.5 References

[**Stallman87**] R. Stallman. *GNU Emacs Manual.* Free Software Foundation, Cambridge, MA, March 1987.

C.6 Button Attributes

We include here a summary of all of the buttons that appear in the 'DEMO' file, generated with the EBut/Help/BufferButs Hyperbole menu item. This information does not appear in that file but is stored separately in a terse format.

```
@loc> "DEMO"

<(factorial)>
  Evaluates a Lisp expression LISP-EXPR.
   actype:          eval-elisp
   args:            ((message "Factorial of 5 = %d" (* 5 4 3 2)))
   creator:         "rsw@cs.brown.edu"
   create-time:     "Nov 25 05:52:22 EDT 1991"
   modifier:        "rsw"
   mod-time:        "Nov 25 05:54:00 EDT 1991"

<(factorial)>
  Evaluates a Lisp expression LISP-EXPR.
   actype:          eval-elisp
   args:            ((message "Factorial of 5 = %d" (* 5 4 3 2)))
   creator:         "rsw@cs.brown.edu"
   create-time:     "Nov 25 05:52:22 EDT 1991"
   modifier:        "rsw"
   mod-time:        "Nov 25 05:54:00 EDT 1991"

<(Info-directory-list)>
  Evaluates a Lisp expression LISP-EXPR.
   actype:          eval-elisp
   args:            ((describe-variable 'Info-directory-list))
   creator:         "rsw@cs.brown.edu"
   create-time:     "Nov 25 23:46:16 EDT 1991"
   modifier:        "rsw"
   mod-time:        "Dec 30 03:17:52 EDT 1991"

<(Info-directory)>
  Evaluates a Lisp expression LISP-EXPR.
   actype:          eval-elisp
   args:            ((message "Info-directory = %s" Info-directory))
   creator:         "rsw@cs.brown.edu"
   create-time:     "Nov 25 23:46:54 EDT 1991"

<(keyboard macros)>
  Displays an Info NODE in another window.
   actype:          link-to-Info-node
   args:            ("(emacs)Keyboard Macros")
   creator:         "rsw@cs.brown.edu"
   create-time:     "Nov 25 21:08:12 EDT 1991"

<(glossary)>
  Displays an Info NODE in another window.
```

```
   actype:          link-to-Info-node
   args:            ("(hypb.info)Glossary")
   creator:         "rsw@cs.brown.edu"
   create-time:     "Dec 10 03:11:29 EDT 1991"

<(keyboard macro)>
  Executes KBD-MACRO REPEAT-COUNT times.
   actype:          exec-kbd-macro
   args:            ("^S(^F^[xdescribe-function^M^M^U^" 1)
   creator:         "rsw@cs.brown.edu"
   create-time:     "Nov 25 20:38:11 EDT 1991"
   modifier:        "rsw"
   mod-time:        "Apr 10 18:34:06 EDT 1992"

<(maximum length)>
  Evaluates a Lisp expression LISP-EXPR.
   actype:          eval-elisp
   args:            ((message "Max length of explicit button labels = %d char-
acters." ebut:max-len))
   creator:         "rsw@cs.brown.edu"
   create-time:     "Nov 25 20:48:26 EDT 1991"

<(shell command)>
  Executes a SHELL-CMD string asynchronously.
   actype:          exec-shell-cmd
   args:            ("ls -l DEMO" nil nil)
   creator:         "rsw@cs.brown.edu"
   create-time:     "Nov 25 20:42:32 EDT 1991"
   modifier:        "rsw"
   mod-time:        "Dec 13 15:07:43 EDT 1991"

<(.login)>
  Displays a PATH in another window scrolled to optional POINT.
   actype:          link-to-file
   args:            ("/u/rsw/.login")
   creator:         "rsw@cs.brown.edu"
   create-time:     "Nov 26 01:01:08 EDT 1991"
   modifier:        "rsw"
   mod-time:        "Dec 13 15:12:02 EDT 1991"

<(tmp directory)>
  Displays a DIRECTORY in Dired mode in another window.
   actype:          link-to-directory
   args:            ("/tmp")
   creator:         "rsw@cs.brown.edu"
   create-time:     "Nov 26 00:03:37 EDT 1991"

<(factorial alias)>
  Performs action given by another button, specified by KEY and KEY-FILE.
   actype:          link-to-ebut
```

```
    args:            ("./DEMO" "factorial")
    creator:         "rsw@cs.brown.edu"
    create-time:     "Nov 25 20:54:30 EDT 1991"
    modifier:        "rsw"
    mod-time:        "Nov 25 20:56:10 EDT 1991"

<(factorial)>
  Evaluates a Lisp expression LISP-EXPR.
    actype:          eval-elisp
    args:            ((message "Factorial of 5 = %d" (* 5 4 3 2)))
    creator:         "rsw@cs.brown.edu"
    create-time:     "Nov 25 05:52:22 EDT 1991"
    modifier:        "rsw"
    mod-time:        "Nov 25 05:54:00 EDT 1991"
```

Appendix D Sendmail Task Support Configuration

Included below is an example for UNIX system administrators of how to configure the '`sendmail.cf`' file of the Sendmail MTA so that it can properly deliver task-based addressed mail.

The example assumes that all task addresses are of the form:

```
<user.task>
```

or

```
<user.task.subtask>
```

All such mail is delivered to `<user>` if that is a valid local address.

The only change necessary is to add something like the following to the beginning of ruleset zero in the '`sendmail.cf`' configuration file:

```
##########################################################################
S0
# On entry, the address has been canonicalized and focused by ruleset 3.
# Handle special cases.....
#
R@                      $#local $:$n                    handle <> form

# Resolve the local hostname to "LOCAL".
R$*<$*$=w.LOCAL>$*      $1<$2LOCAL>$4                   thishost.LOCAL
R$*<$*$=w.uucp>$*       $1<$2LOCAL>$4                   thishost.uucp

# Handle personalized addresses for a set of local users.
CW @ %
# List users on following line who may use task (personalized) addresses.
CP rsw
# To enable for all users instead, substitute '$-' for all '$=P' below.
# If you want to use some other character than . as the task separator
# character, change its usage below.
R$=P.$*<@$j>$*          $#local $:$1                    user.anything@domain
R$=P.$*<@$+.$j>$*       $#local $:$1                    user.anything@host.dom
R$=P.$*<@LOCAL>         $#local $:$1                    user.token@local
R$=P.$~W                $#local $:$1                    user.token
R$=P.$~W.$~W            $#local $:$1                    user.token.token2
##########################################################################
```

Once installed, to test this setup, you can use the following script and just examine the output to see if addresses were parsed to your satisfaction. You should first change all occurrences of `cs.brown.edu` in the script to your own domain.

```
#!/bin/csh -f
#
# Substitute the name of the sendmail configuration file you want to
# test for sendmail.cf below.  If it is not in the current directory,
# use a full pathname.
#
# Also substitute your domain name for all examples of cs.brown.edu below.
```

```
#
# Then run the script like: script > /tmp/tmp
# And look at the output in /tmp/tmp.
#
/usr/lib/sendmail -bt -Csendmail.cf -oQqueue << 'EOF'
3,0,4 root.a,root.role1.role2,root.,root.@cs.brown.edu,
3,0,4 root.r1.r2@cs.brown.edu@mit.edu,root.r1.r2@cs
3,0,4 root.r1@bu.edu
'EOF'
#
# End of script
```

Short Contents