

**PENPOINT™ OPERATING SYSTEM**

# **PenPoint™ UI Design Guidelines**

## PenPoint™ User Interface Design Guide

GO Corporation

Foster City, California

Copyright © 1991 GO Corporation. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language in any form by any means without the written permission of GO Corporation, 950 Tower Lane, Suite 1400, Foster City, CA 94404.

Printed in the United States of America.

Your license agreement with GO Corporation, which is included with the product, specifies the permitted and prohibited uses of the PenPoint™ operating system. Any unauthorized duplication or use of the PenPoint operating system in whole or in part is forbidden. Information in this document is subject to change without notice and does not represent a commitment on the part of GO Corporation.

The following are trademarks of GO Corporation: GO, the PenPoint logo, the GO logo, ImagePoint, PenPoint, GrafPaper, TableServer, BaseStation, EDA, DrawingPaper, and ChartPaper.

Words are checked against the 77,000 word Proximity/Merriam-Webster Linguibase, © 1983 Merriam-Webster. © 1983. All rights reserved, Proximity Technology, Inc.

The spelling portion of this product is based on spelling and thesaurus technology from Franklin Electronic publishers. All other products or services mentioned in this document are identified by the trademarks or service marks of their respective companies or organizations.

### **Warranty Disclaimer and Limitation of Liability**

**GO Corporation makes no warranties, express or implied, including without limitation the implied warranties of merchantability, fitness for a particular purpose and noninfringement, regarding PenPoint software or anything else. GO Corporation does not warrant, guarantee, or make any representations regarding the use or the results of the use of the PenPoint software, other products, or documentation in terms of its correctness, accuracy, reliability, currentness, or otherwise. The entire risk as to the results and performance of the PenPoint software, and documentation is assumed by you. The exclusion of implied warranties is not permitted by some states. The above exclusion may not apply to you.**

**In no event will GO Corporation, its directors, officers, employees, or agents be liable to you for any consequential, incidental, or indirect damages (including damages for loss of business profits, business interruption, loss of business information, cost of procurement of substitute goods or technology, and the like) arising out of the use or inability to use the PenPoint software, other products, and documentation or defects therein even if GO Corporation has been advised of the possibility of such damages, whether under theory of contract, tort (including negligence), products liability, or otherwise. Because some states do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitations may not apply to you. GO Corporation's liability to you for actual damages from any cause whatsoever, and regardless of the form of the action (whether in contract, tort (including negligence), product liability or otherwise), will be limited to \$50.**

### **Note**

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instruction manual may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference, in which case the user will be required to correct the interference at his or her own expense.

# Table of Contents

Preface .....	xi
<b>Section I: Introduction to the Notebook User Interface .....</b>	<b>1</b>
<b>Chapter 1: Designing for Mobile, Pen-Based Computing .....</b>	<b>3</b>
PenPoint Innovations .....	4
Designing for the Notebook Metaphor .....	5
Designing for the Pen .....	6
Designing for the Embedded Document Architecture .....	7
Designing for Scalability .....	8
Designing for Consistency .....	9
A Design Challenge for the '90s .....	10
<b>Chapter 2: The Notebook.....</b>	<b>11</b>
The Notebook .....	12
Fitting Into the Notebook .....	13
<b>Chapter 3: Gestures.....</b>	<b>15</b>
Dual Command Path: Gestures and Controls .....	16
Core Gestures.....	17
Non-Core Gestures .....	18
Letter Accelerators .....	18
<b>Section II: User Interface Building Blocks .....</b>	<b>19</b>
<b>Chapter 4: Controls .....</b>	<b>21</b>
Buttons .....	22
Raised Buttons on Grey Background .....	22
Outlined Buttons on White Background .....	23
Half-Outlined Buttons on White Background .....	23
Square Buttons with Long Labels .....	24
Unadorned Buttons in Menus .....	25
Other Button Styles .....	25

Table of Contents

Lists .....	26
Checklists.....	26
Pop-up Checklists .....	27
Multiple Checklists .....	27
Checkboxes.....	28
Checklists Containing Fields .....	28
Boxed Lists .....	29
Scroll Margins .....	30
Vertical Scroll Margins .....	30
Horizontal Scroll Margins.....	30
Scroll Margin Components .....	31
Scroll Margin Operations .....	31
Scroll Margin Borders .....	32
Scrolling by Flicking .....	32
Scrolling Lists.....	33
Scrolling Checklists .....	33
Scrolling Pop-up Checklists.....	34
Scrolling Multiple Checklists.....	35
Editable Lists.....	36
Gesture Accelerators for Scrolling .....	36
Text Fields .....	37
Overwrite Fields.....	37
Fill-in Fields.....	38
Pop-up Text Sheets.....	39
Embedded Text Boxes .....	40
<b>Chapter 5: Menus .....</b>	<b>41</b>
Controls in Menus.....	42
Buttons, Lists and Fields in Menus.....	42
Multiple Types of Control in a Single Menu .....	43
Two-state Switches in Menus .....	43
Menu Labels.....	45
Menu Behavior .....	46
Input Behavior .....	46
Choosing From Menus .....	46
Menu Layout.....	47
Multi-column Menus .....	47
Separating Groups of Controls.....	47
Hierarchical Menus.....	48



<b>Chapter 6: Dialog Sheets and Option Sheets</b> .....	<b>49</b>
Dialog Sheets.....	50
Option Sheets.....	51
Role of Option Sheets in PenPoint.....	51
Option Sheets for Different Types of Objects.....	52
Invoking Option Sheets.....	52
Dynamic Behavior of Modeless Option Sheets.....	52
Response to Checkmark Gesture.....	52
Relationship to the Selection.....	53
Clean and Dirty Controls.....	53
Copying Options from One Object to Another.....	53
Modeless and Modal Sheets.....	54
Command Buttons.....	55
Standard Modeless Buttons.....	55
Standard Modal Buttons.....	56
Non-standard Buttons.....	56
Command Buttons and Other Buttons.....	57
Layout of Controls.....	58
Standard Layout.....	58
Non-standard layouts.....	58
Single vs. Multiple Sheets.....	59
Pop-up vs. Inline Choices.....	60
Fill-in vs. Overwrite Fields.....	61
De-activating vs. Hiding Controls.....	61
<b>Chapter 7: Status, Warning and Error Feedback</b> .....	<b>63</b>
Busy Clock.....	64
Busy Clock Location.....	64
Wording Guidelines.....	66
Wording of Messages.....	66
Wording of Button Labels.....	66
Progress and Completion Messages.....	66
Pop-up Progress/Completion Note.....	67
In-Line Progress/Completion Message.....	68
Confirmation Notes.....	69
Error Notes.....	70
Timing-Triggered Notes.....	71
Audible Feedback for Warning and Errors.....	72
Message Lines.....	73

<b>Chapter 8: Putting the Building Blocks Together .....</b>	<b>75</b>
Basic Guidelines .....	76
Dual Command Path — Controls and Gestures .....	76
Layering .....	76
User Configurability .....	77
Where to Put It: Menu, Option Sheet, or Palette? .....	78
When to Depart from the Standard Building Blocks .....	80
<b>Section III: Standard User Interface Elements .....</b>	<b>81</b>
<b>Chapter 9: Standard Menus and Commands .....</b>	<b>83</b>
Standard Menus .....	84
Document Menu .....	85
Customizing the Document Menu .....	85
Edit Menu .....	86
Customizing the Edit Menu .....	87
Recommended Menus .....	89
View Menu .....	89
Options Menu .....	90
<b>Chapter 10: Standard Option Sheets .....</b>	<b>91</b>
Accessing The Application Option Sheets .....	92
Title & Info Sheet .....	92
Access Sheet .....	93
Application Sheet .....	94
Adding Your Own Application Option Sheets .....	95
<b>Chapter 11: Icons .....</b>	<b>97</b>
Application Icons .....	98
Icons for Documents .....	99
Icons for Tools .....	99
Icon Design Guidelines .....	100
Simple Shapes .....	100
Lightweight Look .....	100
Feedback When the Icon is Open .....	101
Using Icons to Show Application State .....	102
<b>Chapter 12: Help .....</b>	<b>103</b>
Quick Help .....	104
Help Notebook .....	105

<b>Section IV: Processing Pen Input</b> .....	<b>107</b>
<b>Chapter 13: Processing Gestures</b> .....	<b>109</b>
Gesture Targeting.....	110
The Target Point .....	110
Targetting Unselected Objects.....	111
Targetting the Selection.....	111
Targetting Window Objects.....	112
Core Gestures.....	113
Non-Core Gestures .....	114
Avoiding Gesture Collisions .....	115
Passing on Unused Gestures .....	118
<b>Chapter 14: Processing Handwriting</b> .....	<b>119</b>
Handwriting Translators .....	120
Constraining Translation .....	121
The Problem of Ambiguity .....	121
Improving Translation via Context-Specific Constraints .....	121
Constraining to Letters and Numbers .....	122
Constraining Via Dictionary and Templates.....	122
Guidelines for Using Constraints .....	123
Three Models of Translators and Fields .....	123
<b>Chapter 15: Processing Strokes Without Translating</b> .....	<b>125</b>
<b>Chapter 16: Presenting Input Modes</b> .....	<b>127</b>
Translation Modes .....	128
Always Show the Current Mode .....	128
Mode Control on Menu Line .....	129
Pop-up List at Right of Menu Line .....	129
Palette at Right of Menu Line.....	130
Mode Control in Palette Line .....	131
Mode Control on Pop-up Sheet.....	132
<b>Chapter 17: Handling Keyboard Input</b> .....	<b>133</b>

<b>Section V: Manipulating Application Objects</b> .....	<b>135</b>
<b>Chapter 18: Selection</b> .....	<b>137</b>
User Model .....	138
Selection Feedback .....	138
Selecting and Deselecting.....	139
Selecting a Single Object.....	139
Deselecting an Existing Selection.....	140
Extending a Selection by Dragging .....	141
Extending a Selection With Brackets .....	142
Discontiguous Multiple Selections .....	144
Auto-selection of Gesture Targets .....	144
Text Selection Issues .....	145
Pending Delete.....	145
Primary Input Preference.....	145
Selection in Option Sheets.....	146
<b>Chapter 19: Move/Copy</b> .....	<b>147</b>
User Model: Drag and Drop .....	148
Move and Copy in Mouse-based Interfaces .....	148
Move and Copy in PenPoint.....	149
Support for Implementing Move and Copy .....	150
Initiating the Move or Copy Operation .....	150
Move and Copy Marquee .....	150
Drag Icons .....	151
Completing the Operation .....	153
Drag Rectangle .....	153
Targetting the Destination .....	154
Cancelling the Drag .....	155
Moving and Copying Between Applications.....	156
Standard Data Types .....	156
Object Data Transfer .....	156
Regions that Don't Accept Embeddees .....	157
Examples .....	157
Example 1: Moving a Word in Text.....	158
Example 2: Moving an Item in a List.....	159
Example 3: Moving a Figure.....	160
Example 4: Moving a Figure to a Far Destination .....	161
Example 5: Object Data Type: Copying a Paragraph to the Bookshelf .....	162

<b>Chapter 20: Editing Text .....</b>	<b>165</b>
Pop-up Edit Pads.....	166
Usage by PenPoint .....	166
Usage by Applications.....	166
Invoking Edit Pads.....	167
Input Behavior .....	167
Size Preference.....	167
Using Edit Pads .....	168
Pop-up Writing Pads .....	169
Gestures in Character Boxes .....	170
Gestures in Fill-in Fields .....	171
Gestures in Text Views.....	172
<b>Chapter 21: Forms.....</b>	<b>175</b>
 <b>Section VI: Application Architecture .....</b>	<b>177</b>
 <b>Chapter 22: Documents .....</b>	<b>179</b>
When to Structure Your Application as a Document .....	180
User Model: Document on Single Notebook Page.....	180
Basic Document Layout .....	181
Local Page Controls .....	183
Managing Embedded Objects .....	185
Transparent Embeddors.....	185
Child-aware Embeddors .....	186
 <b>Chapter 23: Tools.....</b>	<b>187</b>
 <b>Chapter 24: Component/Application Model.....</b>	<b>189</b>
 <b>Chapter 25: Notebooks and Notebook Sections.....</b>	<b>191</b>
Structuring Your Application as a Section .....	192
Subclassing the Table of Contents .....	193
Subclassing to Provide Inter-document Functionality .....	193
Subclassing to Provide a Customized User Interface .....	193
Structuring Your Application as a Notebook .....	194
 <b>Chapter 26: Services .....</b>	<b>195</b>
 <b>Chapter 27: Installation of Applications and Resources .....</b>	<b>197</b>



# Preface

## Audience

This book is addressed to anyone who is developing — or considering developing — a PenPoint application.

## Scope of This Book

This book covers a wide range of issues that you will encounter in the course of designing a PenPoint application.

The purpose of providing such a design guide is to ensure a high level of consistency and usability throughout the PenPoint environment.

Some of the guidelines cover topics that will be familiar to anyone familiar with other graphical user interfaces — components such as buttons, choice lists, and menus.

Some of the guidelines are unique to PenPoint — guidelines for using gestures, for example.

**Note:** The final version of this book will contain references to the relevant programmer's documentation for each design topic.

## Toolkit Support

Often the conventions described in this book are quite precise — for example, the layout for controls on option sheets. And often they involve dynamic interactions that are complex and subtle — for example, the dynamic behavior of option sheets.

When reading such specific guidelines, you may find yourself thinking “this is all well and good, but how much work is this going to be for me to support?”

The PenPoint User Interface Toolkit has been designed to support these guidelines. In many cases, the guidelines are implemented by the default behavior of the toolkit. In other cases, a protocol is provided that you must respond to in order to implement the standard behavior. In other cases, you need to combine standard components to obtain the desired behavior.

In any case, you will never be “on your own” in the sense of having to implement the recommended behavior “from scratch.”

## **How to Use This Book**

It will be useful to familiarize yourself with this book at the very beginning of your design process — skimming all the chapters, and reading some parts through.

Then, as you actually get into designing your application, no matter what techniques you use — making sketches, storyboarding, prototyping, etc. — use this book as a companion to the programmer's documentation, referring to various sections as needed.

The book is divided into sections, as follows:

- I. Introduction To The Notebook User Interface** gives a brief orientation to designing applications for mobile, pen-based computing. It introduces the Notebook User Interface, discusses basic concepts such as the Notebook and gestures.
- II. User Interface Building Blocks** describes the set of components available to you to build up your interface. These building blocks include controls such as buttons, fields and lists, and various containers for controls such as menus, dialog sheets, option sheets, and notes.
- III. Standard User Interface Elements** describes the standard user interface elements that are implemented by the PenPoint Application Framework, and how to tailor those standard elements for your application.
- IV. Processing Pen Input** covers issues in gesture processing, handwriting translation, and the use of pen strokes without translating. It also covers how to present input modes to the user.
- V. Manipulating Application Objects** presents detailed guidelines for selecting, moving, copying and editing application objects.
- VI. Application Architecture** presents guidelines on the overall structure of the application and how it is presented to the user. Topics include: structuring applications as documents within a notebook, as pop-up accessories, or as separate notebooks, standard menus and option sheets, conventions for handling application modes, etc.



## Related Documentation

This book is not a user's guide — it will not give you an overview of the features of PenPoint or tell you how to use PenPoint. For documentation on how to use PenPoint, see:

- *Getting Started* A brief introduction to using PenPoint.
- *Using PenPoint* A more complete guide to using PenPoint.
- *Connecting to a Personal Computer* A guide to using a PenPoint computer with other personal computers.

Neither is this book a programmer's guide — it will not tell you how to write a PenPoint application. For that, see:

- *PenPoint Application Development Guide* A variety of guide material for the PenPoint developer covering: SDK (Software Development Kit) installation, PenPoint architecture, writing applications, software development tools, and sample code.
- *PenPoint Architectural Reference, Volumes I and II* Detailed information on each of the subsystems in PenPoint.
- *PenPoint API Reference Manual* "Datasheets" on all the functions and messages comprising the API (Application Programming Interface) to the PenPoint operating system.

## **Current Status and Review Cycle**

This is the first draft, released with the PenPoint Developer's Kit on February 15, 1991.

While many of the chapters are fairly complete, several chapters are basically placeholders at this point. These chapters are clearly marked.

There will be at least one more complete draft of this book circulated throughout the PenPoint developer community before final publication in Fall of '91.

PenPoint itself has been over three years in development. During that time we have learned a great deal about designing for a pen-based environment.

At the same time, the release of the PenPoint Developer's Release is just a start. The promise of mobile, pen-based computing won't be fully realized until there is a thriving market of PenPoint applications.

In that context, we invite your feedback and comments.

To help facilitate the review process, there is a section at the end of each chapter giving issues.

Send your comments to:

Tony Hoerber  
User Interface Coordinator  
GO Corporation  
950 Tower Lane  
14th Floor  
Foster City, CA 94404

Fax: (415) 345-9833  
Phone: (415) 345-7400

# **Section I: Introduction to the Notebook User Interface**

This section presents a brief introduction to PenPoint, and presents some overall guidelines for developing PenPoint applications.



# **Chapter 1: Designing for Mobile, Pen-Based Computing**

Designing for a mobile, pen-based application is not the same as designing a desk-bound, mouse-based application.

This chapter describes some of the key PenPoint innovations, and describes their implications for application design.

## **PenPoint Innovations**

The PenPoint operating system has been designed expressly to meet the needs of the mobile, pen-based computing market.

The key innovations of PenPoint include:

- *Notebook User Interface (NUI)*. A simple organizing metaphor — the Notebook — combined with a document model, commands (called *gestures*) issued with a pen, and powerful handwriting recognition.
- *Embedded Document Architecture (EDA)*. PenPoint's EDA lets the user embed live, editable documents within one another and create hyperlink buttons for navigating through the Notebook.
- *Mobile Connectivity*. Instant-on, detachable networking and deferred I/O make possible truly portable computers for mobile users.
- *Scalable*. While expressly designed for small, lightweight, portable computers, PenPoint is highly hardware-independent and scales to a variety of formats, from pocket-size to wallboard-size computers.

Taken together, these innovations point to a different way of thinking about computers, and a different way of thinking about application design.

The sections that follow suggest some of the implications of these features for application design.

## **Designing for the Notebook Metaphor**

The paper-like world suggested by the notebook metaphor is rich in design possibilities. We are all familiar with notebooks, forms, sheets, pads, tabs, bookmarks, sticky notes, etc. And the graphical conventions used in this world are equally rich: borders, margins, grids, lists, checkboxes, decorations of every stripe.

As far as possible, all the elements of the Notebook User Interface have been designed to make visual and conceptual sense in the context this paper-like world. For example, PenPoint has:

- *A Table of Contents* instead of a *directory of files*.
- *Notebook pages* instead of *application windows*.
- *Turning to pages in the Notebook* instead of *launching/quitting applications and opening/saving files*.
- *Sheets* instead of *control panels*.
- *Notes* instead of *alerts* from the system.
- *Writing pads* instead of *data entry areas*.
- *Menu lines* instead of *menu bars*.
- *Scroll margins* instead of *scrollbars*.
- *Checklists* instead of *radio buttons*.

By designing within the notebook metaphor you will help make the overall PenPoint environment visually and conceptually coherent.

In particular, unless you are deliberately emulating a calculator, stereo panel or some other type of control panel, avoid the "gadget" look with raised buttons and switches, etched lines, heavy shadows, and other visual embellishments.

Of course the notebook metaphor can only be carried so far. For example, buttons are an important element of the Notebook User Interface, and no one ever tapped a button on a real document to make something happen.

But the magic of a powerful metaphor lies not in its precision, but in its ability to bring together different worlds in a useful way. The Notebook User Interface brings together the world of paper, pencils, notebooks and writing, with the world of computers, with its buttons, icons, links, and automatic commands.

## **Designing for the Pen**

The pen is clearly a different sort of input device than the mouse and keyboard combination.

- The pen is a single, simple device that allows the user to both point at objects on the screen and enter data.
- Where the mouse requires the user to first select an object, then choose the command, the pen allows the user to specify both the operand and the operation with a single gesture.
- Where the keyboard requires an insertion point, and some sort of feedback to mark it, the pen allows the user to write wherever desired, or to make a gesture to indicate where to insert text.
- Where the mouse is a relative input device, requiring a point on the screen as its surrogate, the pen is an absolute device, that does not require a pointer.

All of these differences — and more — need to be taken into account in designing a pen-based application.

But over and above the specific characteristics of the pen as an input device, the real challenge is to design applications that fully exploit the pen. And that requires stepping back and re-thinking traditional approaches.



## **Designing for the Embedded Document Architecture**

The Application Framework implements embedding and hyperlinks transparently for all applications, so you don't have to write special code to implement the basic functionality.

Of course there are some things you should do to take full advantage of these features. For example, many applications will want to be aware of embedded objects for layout purposes. And when the user taps a hyperlink button that is linked to an object within your application, the application should scroll the object into view and select it.

But PenPoint's Embedded Document Architecture has implications far beyond the details of handling embedded objects. It makes possible a modular approach to application design and development, which benefits both the application vendor and the user.

Current PC environments encourage a "monolithic" model of application architecture, all applications tend to converge: a word processor adds support for drawings, tables, and charts, a spreadsheet adds support for word processing and charts, etc. This trend has unfortunate consequences for both the developer and the user. Development and testing cycles are stretched out, and the user is locked into using all the functionality bundled into the monolithic application, like it or not.

With PenPoint's EDA, instead of writing a complex, monolithic application that tries to be all things to all people, you can think in terms of a suite of simpler, more focused applications, knowing that the user can combine them as needed. And you can develop and test each unit of functionality separately.

When designing your application, then, one of the first things to ask is "how can I divide this application up into modules of functionality that the user can combine as needed?"

## **Designing for Scalability**

One aspect of scalability relates to hardware. The entire PenPoint operating system has been designed to run on a full range of hardware devices, from small "shirt pocket" notebooks to wallboard-sized work surfaces.

The other aspect of scalability relates to people. Each user is different, and people will use pen-based computers in a variety of environments. So PenPoint allows the user to change the layout of the system at any time, specifying portrait or landscape orientation, left-handed or right-handed layout, different font sizes and type faces, etc.

The PenPoint user interface toolkit includes a powerful *automatic layout* facility that ensures device independence and scalability. The standard components of the PenPoint user interface make use of this facility, so any menus, option sheets and notes you use in your application are rendered in a device-independent and scalable way.

Any time you handle the rendering of your application's data yourself, you should follow these guidelines:

- Lay out the objects in your application in terms of device-independent *layout units*, rather than absolute units of measure or units for a particular device.
- Express layout relationships in higher-level terms — to the right of, below, etc., whenever possible. This will allow your application to adapt well to different orientations and screen sizes.
- Follow the preference settings for **User Font** and **Font Size**. That way you'll allow the user to adjust for personal preference or ambient lighting conditions.

## **Designing for Consistency**

The graphical user interfaces of the '70s and '80s have shown the value of consistency in user interface design. As pen-based computers extend the use of computers to new groups of people — people who are less technical and have less time to spend learning computer arcana — consistency will become even more important.

PenPoint lays the foundation to support this new class of users by providing consistent interfaces for many commands and options, including some that in traditional environments are presented differently by each application.

Examples include **Print** and **Print Setup**, **Send**, **Find**, **Spell**, **Import** and **Export**, and many others. Of course, you can customize these interfaces if you need to. But in most cases using the standard building block that PenPoint provides will allow you to concentrate on bringing the particular added value of your application to the marketplace.

In addition, PenPoint supports a set of 11 core gestures that all applications should implement where appropriate.

Another example of how PenPoint promotes consistency is the “drag and drop” method of moving and copying objects, which allows the user to move or copy objects of all types — documents, icons, text, figures in a drawing program, etc. — in the same way.

## **A Design Challenge for the '90s**

This chapter has briefly sketched out some of the new design territory that PenPoint opens up.

The rest of this book is a detailed guidebook for exploring that territory. It spells out in detail how to design your application to take full advantage of PenPoint's features — the notebook metaphor, gestures and handwriting recognition, embedding and hyperlinks, mobile connectivity, scalability, and consistency.

Many people today share a vision of a new class of mobile, pen-based computers that bring the power of computers to people in a more natural way. The enabling technologies are now in place to realize that vision: hardware technologies such as compact, lightweight batteries, flat, low-power, readable displays, fast, low-power chips. And now, with PenPoint, the software foundation and framework is in place.

But there is one essential piece missing — perhaps the most important piece. The promise of the new class of mobile computers won't be fully delivered until there is a rich set of applications that put all of the technology to work for people in their daily lives.

That's where you come in.

Just as the introduction of graphic displays and the mouse as an input device created a rich space of design possibilities in the '70s and '80s, the notebook computer and the pen will do the same in the '90s.

We invite you to join us in meeting this exciting challenge.

## **Chapter 2: The Notebook**

This chapter introduces the basic elements of the Notebook User Interface, including the Notebook, the Bookshelf, the Table of Contents, notebook tabs and hyperlinks.

## The Notebook

The Notebook is shown in Figure 1.

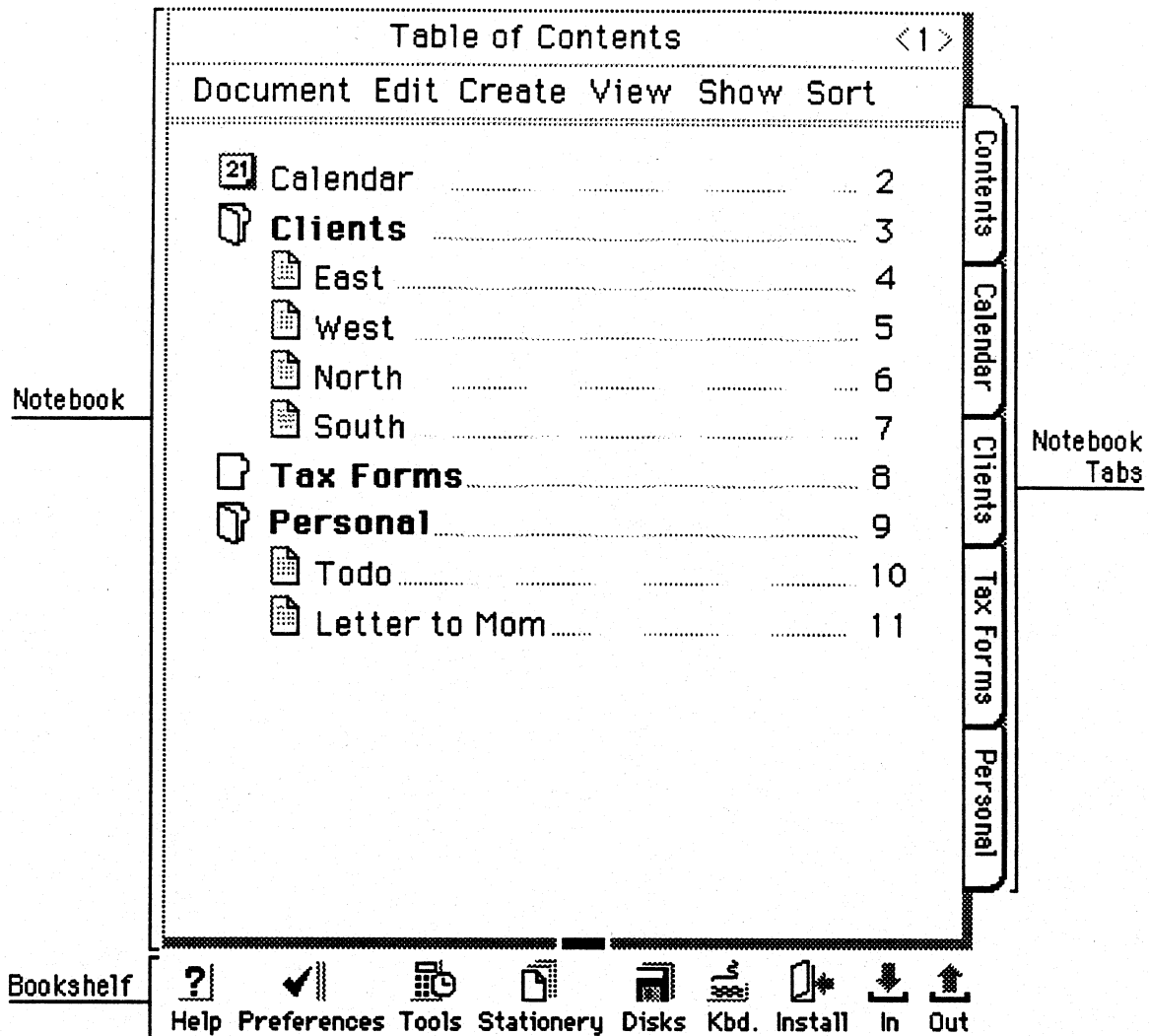


Figure 1: The Notebook and Bookshelf

As shown in Figure 1, the first page of the Notebook contains the Table of Contents. Each page of the Notebook contains either a single document, or a section divider. The Table of Contents shows documents in normal font, sections in bold.

The user can turn to any page by tapping on the page number. The user can also perform housekeeping functions from the Table of Contents, such as moving, copying, renaming and deleting documents.

The user can also access documents by means of notebook *tabs* and *hyperlink buttons* that can link any document to any other document.

Below the Notebook is an area called the *Bookshelf* that contains icons representing utilities such as Help, Preferences, In and Out boxes, etc.

## Fitting Into the Notebook

The most straightforward way to fit into the notebook is to present each application instance as a document on a single page of the notebook.

That is not the only way to present your application, however. Other standard ways of presenting application instances include:

- *Pop-up tool.* Some applications make more sense as a pop-up tool than as a document in a notebook. Examples include utilities such as a clock, a calculator, a snapshot program, etc.
- *Separate Notebook.* You can either use the standard PenPoint notebook as a vehicle for delivering information, or you can tailor the notebook interface to suite the needs of your application.
- *Background Service.* This category includes transfer services that manage the sending and receiving of information through the Outbox and Inbox, and other services such as database servers.

These approaches are described in more detail in Section VI, *Application Architecture*.





## Chapter 3: Gestures

This chapter introduces the concept of using pen gestures to issue commands.

Topics covered include:

- *Dual Command Path.* The basic PenPoint principle of using gestures to complement visible controls.
- *Gesture Categories.* Core gestures, non-core gestures and capital letter accelerators.
- *Core Gestures.* Description of the 11 core gestures.

Gesture processing is discussed in detail in Chapter 13, *Processing Gestures*.

## **Dual Command Path: Gestures and Controls**

It is a good practice in graphical user interfaces to provide the user with some kind of visual cue for the most important commands in any given context.

In PenPoint this means that you should make most of the commands in your application available from some form of visible control — buttons, menus, dialog sheets or option sheets. (These components are described in detail in Section II.)

In addition to the visible controls, you should provide gestures for the most common or important commands.

This *dual command path* is a basic principle of the PenPoint user interface.


Here are a few examples of how gestures and visible controls complement each other in PenPoint. These examples should give an idea of how to apply this idea of the dual command path in your application.


- *Turning notebook pages.* The user can either tap the arrows on either side of the page number at the right of the Title Line, or make a flick gesture anywhere in the Title Line to turn to the next or previous page (flick left for next page, right for previous page.)
- *Scrolling.* The user can either use the arrows or drag box on the scroll margin, or flick directly on the text or list to scroll it (flicking up shoves the lines up, flicking down shoves the lines down).
- *Editing operations.* The common editing operations such as **Delete** and **Options** are available from both the **Edit** menu and from gestures (X for **Delete**, checkmark for **Options**.)
- *Text style options.* Text styles such as **Bold**, **Italic**, **Underlined** and **Normal** are all available from both the text option sheet and from gestures (the first letter of each command: **B**, **I**, **U** and **N**).


The sections that follow discuss the three categories of PenPoint gestures: *core gestures*, *non-core gestures*, and *letter accelerators*.


## Core Gestures


There are 11 core PenPoint gestures:


- 


*Tap.* This is the most basic gesture. Typically tapping selects an object or pushes a button.
- 


*Press-Hold.* (Touch the pen to the screen and pause for a moment.) Initiate drag move when made over icons or selected application objects. Initiate drag area select when made over background or unselected objects in application.
- 


*Tap-Hold.* (Tap the screen once, then touch the pen to the screen again and pause for a moment.) Initiate drag copy.
- 


*Flick.* (Four directions) Flicking is used throughout PenPoint to bring more information into view. The user model is that the flick gesture shoves the object in the direction of the flick. Examples include scrolling text, turning notebook pages, and exposing overlapping tabs.
- 


*X.* The basic deletion gesture.
- 

*Caret.* Insert or create.
- 

*Circle.* Edit.
- 

*Checkmark.* Display the object's option sheet.
- 

*Brackets.* Adjust an existing selection in contexts that support the selection of a span of objects (such as text, lists and tables).
- 

*Pigtail.* Delete a single character in text.
- 

*Down-Right.* Insert a space in text.

These gestures, because they represent operations that are common across all or most applications, are considered fundamental, or core.

Applications are strongly encouraged to use these gestures, and are expected to follow the GO usage if these gestures are used.

## **Non-Core Gestures**

In addition to the core gestures, PenPoint supports a larger set of gestures (approximately 20, depending on how they are counted) that are not universally applicable. Examples include variations on core gestures (e.g. multiple taps and multiple flicks) and other symbols (e.g. arrows, square, and plus). Applications are encouraged to be consistent with GO usage, but we anticipate that there will be some variation in the use of these gestures.

The non-core gestures are described in detail in the chapter on *Processing Gestures*.

## **Letter Accelerators**

In addition to the special shapes in the first two categories, PenPoint allows any capital letter to be used as a gesture. The normal usage is to use the first letter of the command, e.g. **S** for **Spell** or **F** for **Find**.

Because the capital letters are intended to be used as context-specific accelerators, their interpretation will often vary depending on the context. For example, in PenPoint a **B** drawn in text toggles the bold attribute on/off, while a **B** drawn on the title line of a document toggles the document's borders on/off. Another application might well place a third interpretation on **B**.

**Note:** In the Developer's Release of PenPoint, many of the capital letters aren't recognized. This will be fixed for PenPoint 1.0.

## Section II: User Interface Building Blocks

This section describes the building blocks PenPoint provides to build your user interface.

Chapters include:

- *Controls*. The set of standard user interface components such as buttons, lists, and fill-in fields.
- *Menus*. Pop-up menus containing commands and checklists.
- *Dialog Sheets and Option Sheets*. Pop-up sheets for setting parameters to commands or attributes of objects.
- *Status, Warning and Error Feedback*. Various types of user feedback and messages, including the standard busy clock, progress and completion messages, confirmation messages and error messages.
- *Putting the Building Blocks Together*. Which building blocks — menus, option sheets or tiled palettes — to use in various situations



## Chapter 4: Controls

PenPoint provides a rich set of user interface components such as buttons, checkboxes, lists and so on. Collectively these components are referred to as *controls*.

This chapter describes the standard PenPoint controls and gives guidelines for their usage.

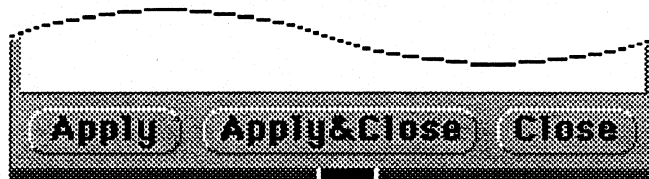
## **Buttons**

The simplest type of control is the *button*, which allows the user to initiate an action.

There are several standard button forms used throughout the interface for PenPoint. The sections that follow illustrate each of these forms.

### *Raised Buttons on Grey Background*

PenPoint uses this form for the action buttons of dialog and option sheets, as shown in Figure 2.



**Figure 2: Raised Buttons on Grey Background**

The use of the white/dark grey outline on the light grey background gives a raised, "push button" effect.

The raised button style is appropriate when you deliberately intend to produce the effect of a mechanical control panel in a physical device such as a calculator, keyboard, tape recorder, etc.

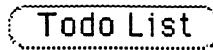
The raised button style should be used sparingly, however, as it produces a strong visual effect that tends to work against the notebook and paper metaphor of PenPoint.

In particular, do not use raised buttons on surfaces that the user would otherwise perceive as flat and paper-like.



### *Outlined Buttons on White Background*

PenPoint uses this form for Hyperlink buttons, as shown in figure 3.



**Figure 3: Outlined Button on White Background**

### *Half-Outlined Buttons on White Background*

The half-outlined button is a special form designed to facilitate the use of buttons in the context of surrounding text.

Figure 4 shows a paragraph of help text with two Hyperlink buttons that the user can tap to see related topics.

PenPoint features two forms of help. You can get Quick Help at any time by simply tapping on the Help icon and then tapping on any object visible on the screen to see a brief message about the object. Or, you can open the Help Notebook and browse through it. The Help Notebook contains information, organized by topic, for both PenPoint itself and for each installed application.

**Figure 4: Half-Outlined Buttons in Text**

The open left end allows the button to be read as part of the flow of the text, while the curved right distinguishes the button from an underline, and provides the suggestion of a button. The absence of the upper border allows for normal line spacing.

Note: To obtain the effect described above you need to explicitly set the label font and size for each button to match the surrounding text.

*Square Buttons with Long Labels*

This form works well for columns of buttons that have long labels.

Figure 5 shows a typical example.

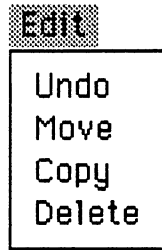
- Lower Case Letters
- Upper Case Letters
- Numerals & Symbols
- Individual Characters
- Exit

**Figure 5: Square Buttons with Long Labels**

### *Unadorned Buttons in Menus*

Unlike buttons in most settings, buttons in menus do not have any type of outline or other ornamentation to set them apart. This is because buttons are the most common type of control in menus, and the layout of menus is simple.

Figure 6 below shows a typical menu with several buttons.



**Figure 6: Unadorned Menu Buttons**

### *Other Button Styles*

You can modify the shape and border style of buttons to obtain visual effects other than those described above by setting various toolkit style bits.

Follow these general guidelines when customizing button styles:

- Don't depart from the standard button styles just to be different. Usually, minor visual variations do not add substantial value to an application — but they do present one more thing for the new user to become familiar with. By using the standard styles, your application will better fit in with the overall PenPoint environment.
- If you do depart from the standard styles, pick a single style and use it consistently. Don't introduce two or three new styles without strong motivation.
- Avoid the temptation to overuse heavy borders and thick shadows. Heavy borders and shadows tend to degrade readability because they draw attention away from the button's label and interact with the letters in the labels to set up competing visual shapes.

## Lists

PenPoint provides several types of controls for the presentation of lists to the user.

### Checklists

Checklists allow the user to choose one item or setting from a list of choices.

Figure 7 shows a typical example.

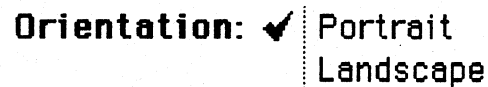


Figure 7: Checklist

By default the list is layed out in a single column. You can specify that the choices be arranged horizontally, or in a matrix.

You can also specify that the individual choices be represented as graphical images rather than as text strings, as shown in Figure 8.

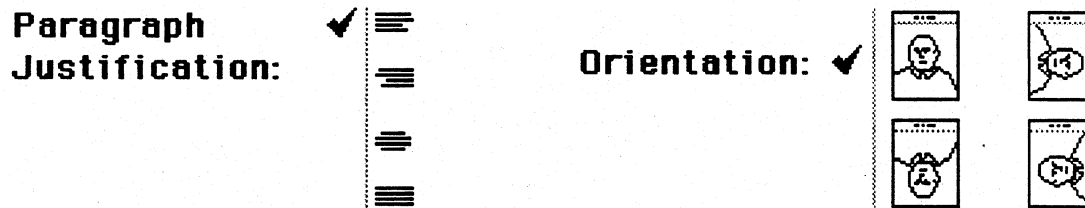


Figure 8: Checklists

Only use graphical images when they communicate more effectively than text. Take the two examples in Figure 8 above. In both cases the images are clear and unambiguous, and the purpose of the control is evident at a glance.

### Pop-up Checklists

Pop-up checklists allow you to present choices in a more compact format.

Figure 9 shows a typical example.



**Figure 9: Pop-up Checklist**

The currently selected choice appears to the right of the list's label, separated by a small arrow, as shown on the left. Tapping on the arrow or the current choice pops up a menu with the entire list, as shown on the right.

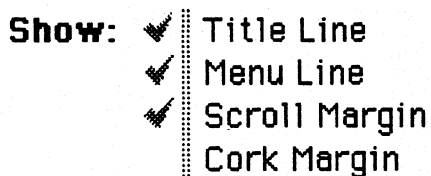
The user can also make the flick and double-flick gestures (on either the arrow or the current choice) to choose from a pop-up checklist in a single step, without bringing up the menu. A flick up or left makes the next item in the list current. Similarly, a flick down or right makes the previous item in the list current. A double-flick up or left makes the last item current, while a double-flick down or right makes the first item current.

### Multiple Checklists

In some lists the choices are not mutually exclusive — more than one choice can be on at a time.

For such situations PenPoint provides a control called a *multiple checklist*, in which tapping on one choice toggles its checkmark on and off, without affecting the other choices in the list.

Figure 10 shows a typical multiple checklist.



**Figure 10: Multiple Checklist**

The user can always tell whether a checklist is single or multiple by looking at the line between the checkmark and the list: a single line signifies a single-choice list, a double line signifies a multiple-choice list.

### Checkboxes

The double-line convention described on the previous page is the standard used by PenPoint to present multiple choice lists in dialog sheets, option sheets and menus.

PenPoint also provides checkboxes for multiple choice lists. Use the checkboxes rather than the double-lined list where appropriate. Situations that call for checkboxes include:

- A single checkbox representing a toggle switch;
- A form where checkboxes are the common convention;
- A columns of checkboxes in a table.

Figure 11 shows how checkboxes in the notebook Table of Contents provide an easy way for the user to toggle the tabs on and off for individual sections and documents.





 <b>Clients</b> .....	3	<input checked="" type="checkbox"/>
 East .....	4	<input type="checkbox"/>
 West .....	5	<input checked="" type="checkbox"/>
 North .....	6	<input checked="" type="checkbox"/>

Figure 11: Checkboxes

### Checklists Containing Fields

Checklists are not constrained to a single type of control.

Figure 12 shows a common idiom, in which one of the choices in a checklist contains one or more overwrite or fill-in fields.

**Pages:**  All  
                   From  to

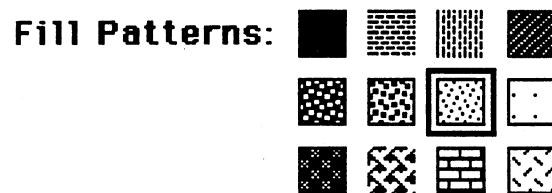
Figure 12: Checklist with Overwrite Fields

### Boxed Lists

An alternative to checklists is the boxed list, in which the current choice is outlined instead of marked with a checkmark.

Use boxed lists when the choices are best represented as graphical images rather than text labels, and when you need to present them in the most compact form possible.

Figure 13 shows a boxed checklist for choosing a fill pattern in a drawing program.



**Figure 13: Boxed List**

As with checklists, you can specify that the choices in the list are mutually exclusive or not.

Figure 14 shows a boxed list from which the user can choose more than one translation mode.



**Figure 14: Multiple Boxed List**

Boxed checklists are the ideal control to use in constructing palettes and control margins, as described in the chapters on *Putting The Building Blocks Together*, and *Presenting Input Modes*.

## Scroll Margins

In order to allow long documents or lists to be displayed in a smaller viewing region, PenPoint provides a control called a *scroll margin*.

### Vertical Scroll Margins

If there is more information past the top or bottom edges of the viewing region, then a *vertical scroll margin* appears on the right edge of the document, as shown in Figure 15.

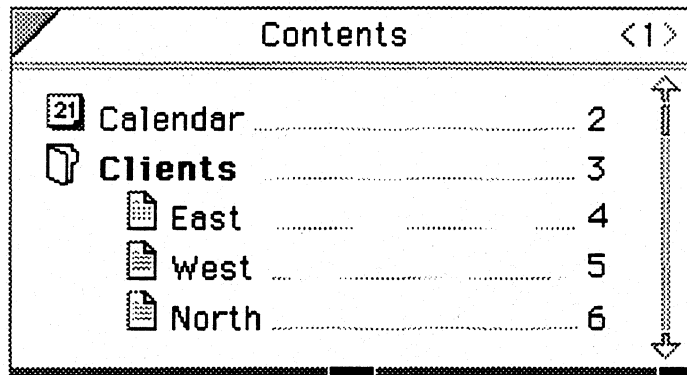


Figure 15: Vertical Scroll Margin

By default, PenPoint displays vertical scroll margins on the right. The user can specify that they be displayed on the left, by means of the Preferences facility.

### Horizontal Scroll Margins

If there is more information past the left or right edge, a *horizontal scroll margin* appears at the bottom of the page. Figure 16 shows a document with both a vertical and a horizontal scroll margin.

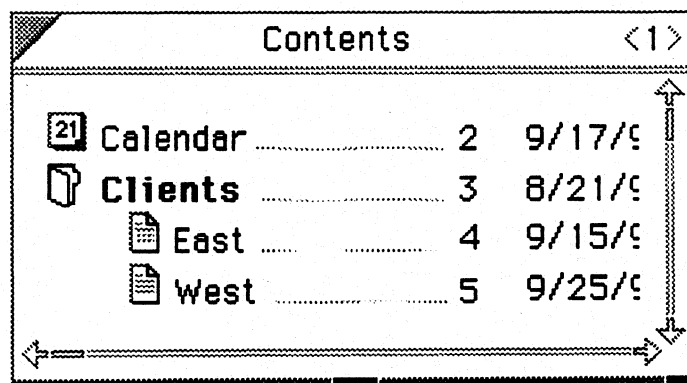
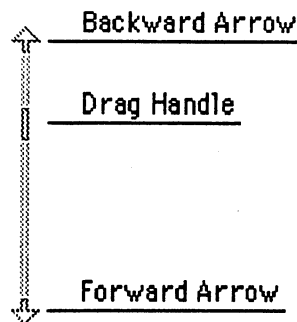


Figure 16: Vertical and Horizontal Scroll Margins



### Scroll Margin Components

Figure 17 shows the components of a vertical scroll margin.



**Figure 17: Scroll Margin Components**

### Scroll Margin Operations

The scroll margins support the following operations:

- *Scrolling by unit.* You should interpret a tap on one of the scrolling arrows as a request by the user to scroll by one unit. In applications that are organized by lines, such as text or lists, the line is the obvious unit. For tables the unit should be a row or column. For painting programs, the unit might be as small as a single pixel.
- *Scrolling by screenful.* A single tap above the handle should scroll towards the beginning of the sheet by one screenful. A tap below the handle should scroll towards the end of the sheet by one screenful.
- *Scrolling to beginning/end.* A double-tap above the handle should scroll all the way to the beginning of the sheet. A double-tap below the handle should all the way to the end of the sheet.
- *Thumbing.* When the user drags the handle of the scroll margin, you should scroll to the corresponding location in the sheet — dragging the handle to the top should result in a scroll to the beginning of the sheet, dragging half-way should result in a scroll to the half-way point of the sheet, etc.

### *Scroll Margin Borders*

By default the scroll margin does not have a border separating it from the body of the scrollable region. This is intentional, so that the double-line appears as a visual separator marking out a margin on a paper-like surface.

For some applications, such as painting programs, it is important that the user be able to see the precise extent of the scrollable region, down to the pixel level. In such cases you can turn on the border of the scroll margin to provide the demarcation.

### *Scrolling by Flicking*

In keeping with the principle of the dual command path, PenPoint provides a family of gestures — known as *flicks* — for scrolling.

The model presented to the user is that he or she is manipulating a sheet of paper directly by shoving it with the pen. Therefore flicking *up* should always move the contents of the sheet *up*, and reveal more information that had been hidden past the *bottom* edge. Likewise, flicking *down* should always move the contents of the sheet *down*, and reveal more information that had been hidden past the *top* edge.

The two most important scrolling gestures are the single and double flicks. In most cases you should interpret them as follows:

- *Single Flick*. Send the line or point under the start of the flick to the edge of the viewing region. So flicking up on a line should send it to the top edge, and flicking down on line should send it to the bottom edge.
- *Double Flick*. Scroll as far as possible. That is, double-flicking up should scroll to the end of the sheet, and double-flicking down should scroll to the beginning of the sheet.

The single and double flicks should suffice for most applications. However, PenPoint also supports the triple and quadruple flick gestures. If your application is organized hierarchically, with more levels of structure, you may want to use the double and triple flick to scroll to intermediate levels, and quadruple flick to scroll all the way to the beginning or end.

## Scrolling Lists

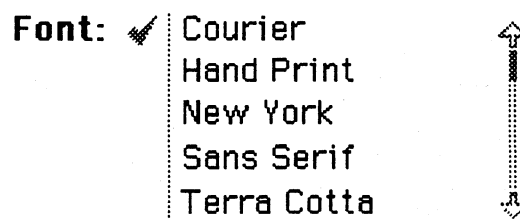
The PenPoint toolkit supports various forms of scrolling lists.

When using a scrolling list, one of the essential design questions to ask is how is the information in the list to be used. There are several usage patterns which suggest different variations, as described below.

### *Scrolling Checklists*

When the purpose is to allow the user to indicate that exactly one of the items is "current," or distinguished for some purpose, the scrolling list behaves like a single choice list control.

Figure 18 shows a scrollable single choice list in the standard PenPoint layout used in dialog and option sheets (bold label followed by an unbordered list.)



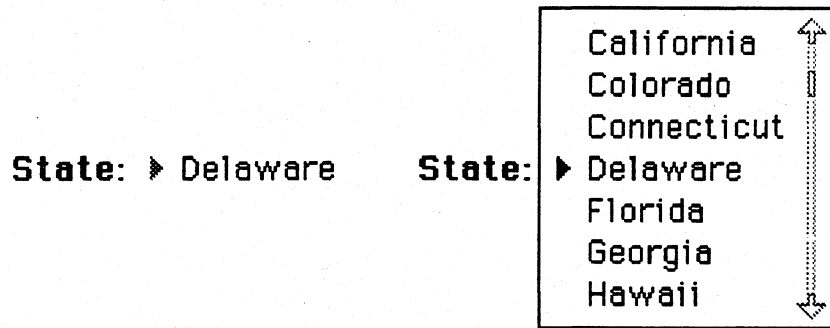
**Figure 18: Scrolling Checklist**

Depending on the context, you may find it more appropriate to present the list in borders and with some other marker for the current item, such as a checkbox.

### *Scrolling Pop-up Checklists*

Use a scrolling pop-up checklist when you need to present choices in a more compact format.

Figure 19 shows a pop-up list from which the user can choose one of the fifty U.S. states.



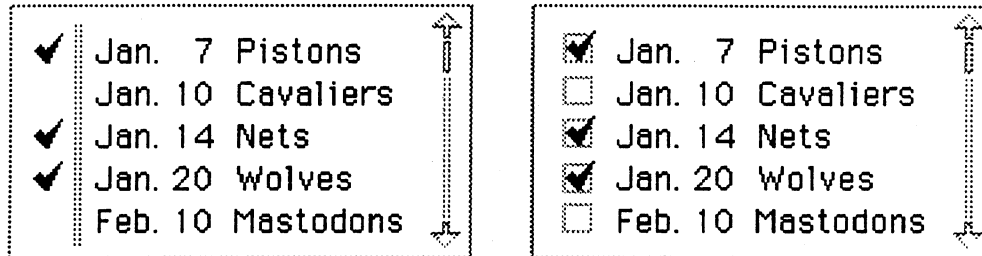
**Figure 19: Scrolling Pop-up Checklist**

The currently selected choice appears to the right of the list's label, separated by a small arrow, as shown at the left in the above figure. Tapping on the arrow or the current choice pops up a menu with the entire list, as shown at the right.

### Scrolling Multiple Checklists

When the purpose is to allow the user to choose more than one of the items in the list, the scrolling list behaves like a multiple choice control.

Figure 20 shows two examples.



**Figure 20: Scrollable Multiple Choice Lists**

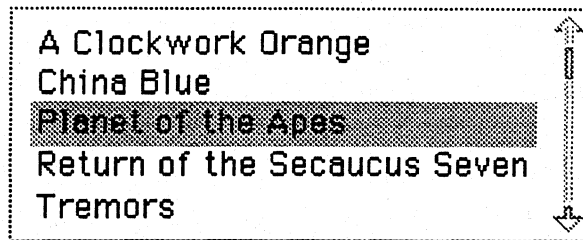
The list on the left uses the PenPoint double-line convention, the list on the right uses checkboxes.

Use the double-line form if the list is in a standard dialog sheet or option sheet. Use checkboxes if you are using checkboxes for similar purposes throughout in your application.

*Editable Lists*

When the contents of the list can be edited by the user (i.e. the user can add, delete and edit items) the list should support selection.

Figure 21 shows a list with one item selected.



**Figure 21: Editable List**

See the chapter on *Selection* for guidelines on handling the selection.

*Gesture Accelerators for Scrolling*

In long alphabetically sorted lists, it is a good idea to use the capital letters as accelerators for scrolling. Allow the user to write the letter anywhere on the list. When a letter is recognized, select the next item in the list beginning with that letter. If the item is not visible, scroll the list so that the item appears at the top of the view.

## Text Fields

There are several types of text fields that are commonly used as controls:

- *Overwrite Fields*. Fields that display each character in a discrete box and allow direct overwrite of characters;
- *Fill-in Fields*. Single-line fields that are backed by pop-up editing pads;
- *Pop-up text view*. Single-line fields that are backed by multi-line, scrollable, fully editable pop-up text sheets.
- *Embedded text view*. Boxes containing scrollable, fully editable text.

This section introduces each of these types of fields. For details on text editing, see Chapter 20, *Editing Text*.

### *Overwrite Fields*

For fields that are limited to a few characters, such as numeric fields, PenPoint provides a type of field that consists of a series of *overwrite boxes*, each of which accepts and displays a single character.

Figure 22 shows a typical usage of overwrite boxes.

**Time:** 1 2 : 3 0

**Figure 22: Overwrite Fields**

Overwrite boxes are designed to make it as convenient as possible to edit the contents of the field. To change a character, the user simply writes the new character directly over an existing one.

Editing in character boxes is described in detail in the Chapter 20, *Editing Text*.

### *Fill-in Fields*

Fill-in fields allow the user to enter textual data with the pen. Figure 23 shows an example.

**Name:** Nathaniel Bowditch

**Figure 23: Fill-in Field**

When fill-in fields are empty, the user can write directly into them. After a short pause, the written characters are translated into a standard computer font.

The user can also tap to pop up an editing pad containing overwrite boxes.

See the chapter on *Editing Text* for details on fields, including pop-up pad behavior and a list of the editing gestures accepted by fill-in fields.

### Visuals

Figure 28 above shows the default field format used in PenPoint dialog and option sheets. The label is at the left, followed by an underlined field.

If your application needs to emulate a standard paper form, you may find it more appropriate to use different ornamentation for fill-in fields, such as boxes around the field, dashes or other characters as separators, etc.

### Font

By default, fill-in fields use the font specified as the **Field Font** preference, and the size specified as the **Font Size** preference.

Your application can override these preferences, either for the purpose of fixing the font and size to ensure that all the information on a sheet can be viewed at once, or because your application provides its own user interface for controlling fonts.



### Pop-up Text Sheets

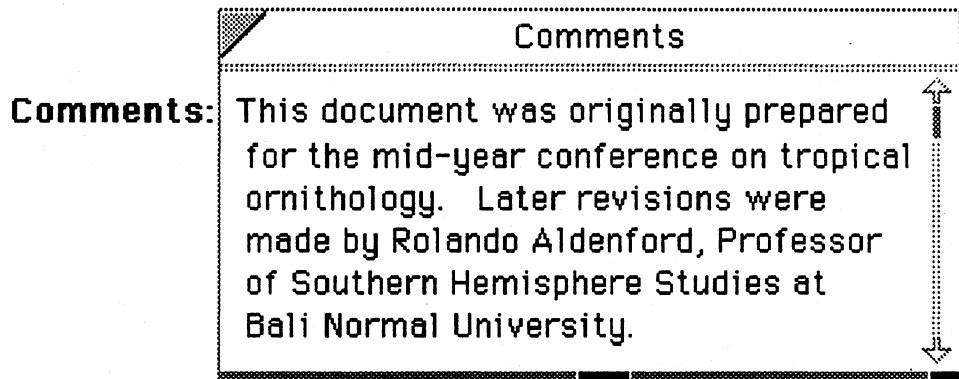
The fill-in fields described above work well for short fields. When the contents of the field will typically be longer than can be displayed in a single-line field, tapping on the field should bring up a pop-up text sheet, instead of an editing pad.

Figure 24 shows a comment field from a document option sheet.

**Comments:** This document was origina...

**Figure 24: Long Text Field**

The ellipsis at the end of the string indicates to the user that the field is truncated. Tapping on the field (or drawing a circle) brings up a pop-up sheet containing a fully-editable text component, as shown in Figure 25.



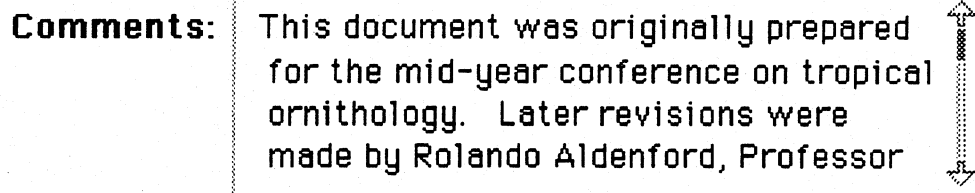
**Figure 25: Pop-up Text Sheet**

The sheet should have the field's label as its title.

### Embedded Text Boxes

If you have room, presenting a long text field as an embedded text box instead of a single-line field backed by a pop-up sheet has the advantage of eliminating the step of popping up the sheet to see the field contents.

Figure 26 shows the comment field implemented as an embedded text box.



**Figure 26: Embedded Text Box**

You can specify that the text box have resize handles on the bottom edge, lower right corner, or right edge, independently.

Often it is most convenient for the user if the width of the box is fixed to be as wide as the document or sheet that contains it, and only the bottom edge has a resize handle, as shown in the example above.

### Issues

Is it important for the toolkit to provide other controls — e.g. gauges and sliders — in the first release?

Applications may want to implement other scrolling models in addition to the methods described in this chapter that are standard in PenPoint. One example of this is a “proportional flicking” model, in which the length of the scroll depends on the length of the flick. Another alternative scrolling model is direct panning via a “hand” mode that is entered via a press-hold timeout.

## Chapter 5: Menus

One of the most basic user interface components is the pop-up menu.

This chapter gives guidelines for menu usage, including:

- The various types of controls that menus can contain
- Conventions for laying out the controls in menus
- Wording conventions for menu labels
- Hierarchical menus

## Controls in Menus

This section describes the controls that menus can contain.

### Buttons, Lists and Fields in Menus

Menus can contain buttons, checklists, multiple checklists, fill-in fields, and overwrite fields, as shown in Figure 27.

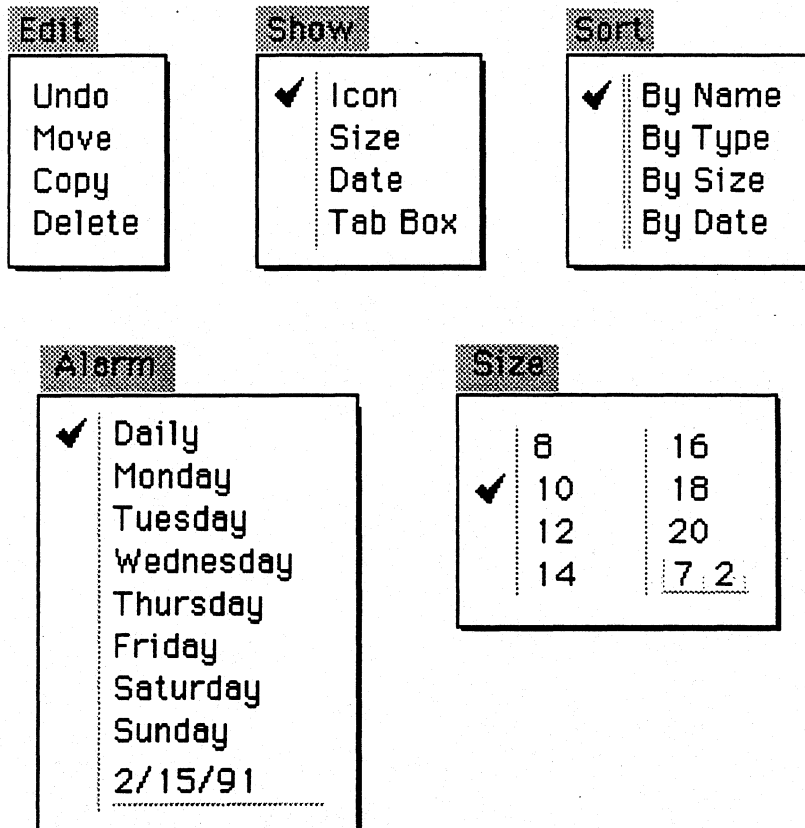
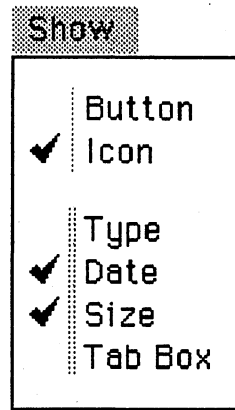


Figure 27: Menus

### Multiple Types of Control in a Single Menu

A single menu can contain more than one type of control, as shown in Figure 28.



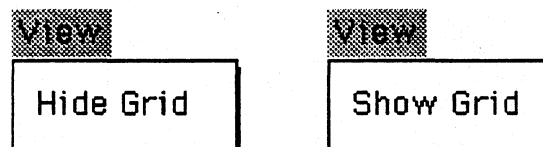
**Figure 28: Menu with Two Kinds of Controls**

### Two-state Switches in Menus

There are two recommended idioms for representing a two-state switch in a menu.

The first is to present the switch in the form of a single command, whose label changes dynamically based on the state of the program.

Use this method when the choices are obvious opposites. A typical example is a command to toggle a grid on and off. The command invites the user to **Hide Grid** when the grid is on, and **Show Grid** when the grid is off. Figure 29 shows the menu in both states.

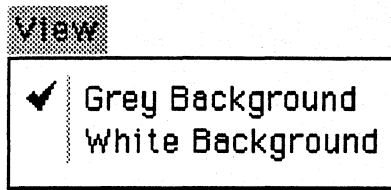


**Figure 29: Single Command for Two-state Switch**

Another example is toggling the label of a single-level undo command between **Undo** and **Redo**.

The other recommended method for presenting a two-state switch is to use a checklist with two items.

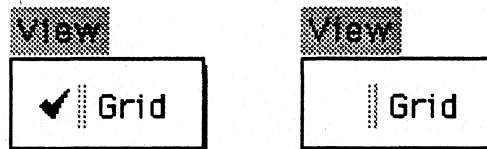
Use this form when the choices do not constitute an obvious pair of opposites. Figure 30 shows a typical example, for setting the color of the background.



**Figure 30: Checklist for Two-State Switch**

Since the choices are not clearly opposites, it is clearer to show them both, rather than having a single command labelled **Set Background to Grey** and **Set Background to White**.

Presenting a two-state switch as a multiple choice list with a single choice, as shown in Figure 31, is not recommended.



**Figure 31: Multiple Checklist for Switch (Not Recommended)**

In this idiom the state of the switch is indicated by the presence or absence of a checkmark. Avoid this usage, as it is not as clear as the two methods described above — a single command whose label toggles, or a checklist showing both choices.

## Menu Labels

The following conventions have become de-facto standards for graphical user interfaces. Use them in PenPoint too.

- *Use Verbs.* When possible, use verbs for both the label of the menu and the commands contained in the menu.
- *Capitalize.* Capitalize the first letter in each word of both label and contents of the menu.
- *Ellipsis convention.* If the menu command will bring up a pop-up sheet, add an ellipsis (three dots) to the label.
- *De-activating menu labels.* If a menu control is unavailable for any reason, set it to the *inactive* state. The toolkit will render it in light grey. In particular, before displaying any menu containing commands that act on the selection, make sure to check whether your application currently holds the selection. If it doesn't, de-activate the selection-based commands before displaying the menu.

## Menu Behavior

PenPoint menus follow the "tap-tap" usage model rather than the "press-drag-release" model. That is, the user taps on the menu's label to display the menu, then taps on a control in the menu to take the desired action and dismiss the menu.

### Input Behavior

Menus are modal: while a menu is displayed, input to the rest of the screen is blocked.

Tapping anywhere on the screen outside the menu dismisses the menu and cancels the mod without taking any other action.

PenPoint makes a deliberate exception to this rule for the other menu labels on the screen. They remain active, so that the user can easily browse through a group of menus by tapping each label, without needing to dismiss the previously-opened menu each time.

### Choosing From Menus

In most cases, tapping on a control in the menu takes the appropriate action and dismisses the menu.

PenPoint makes an exception to this rule when the menu contains a choice list with a fill-in or overwrite field, as shown in Figure 32.

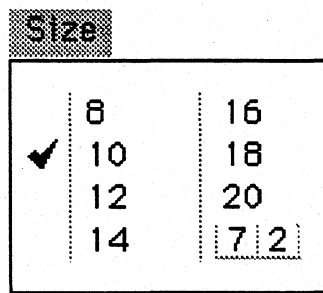


Figure 32: Menu Containing Overwrite Field

Tapping to the left of the vertical line, by the field, will choose that item and dismiss the menu, as usual. But directing input to the field itself (entering a character, making an editing gesture, or tapping to set the keyboard input focus) will *not* dismiss the menu. This allows the user to complete the editing operation and verify that the contents of the field are correct before dismissing the menu.

Similarly, tapping on a fill-in field in a menu displays the field's edit pad, and dismissing the pad leaves the menu displayed.



## Menu Layout

The most readable and most common layout for controls in menus is a single column.

### *Multi-column Menus*

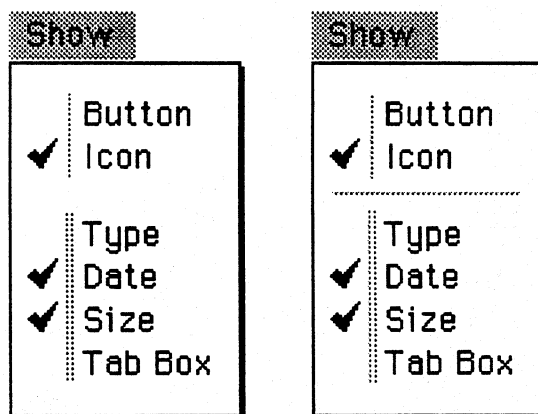
In some cases the standard single column layout results in an extremely tall and skinny menu. An example might be a menu containing the numbers 1 through 20. In such cases consider using a multi-column layout. The rule of thumb is to keep the menu a bit taller than wide — say a 5 to 3 ratio.

For readability, use a column-major sort order when the controls are sorted alphabetically or numerically. That is, the controls should read downward, in columns, rather than across, in rows.

### *Separating Groups of Controls*

You can separate groups of commands either with white space or with a light grey line.

Menus that contain more than one checklist don't need a dividing line, since the vertical single or double line unifies each list, as shown in Figure 33.

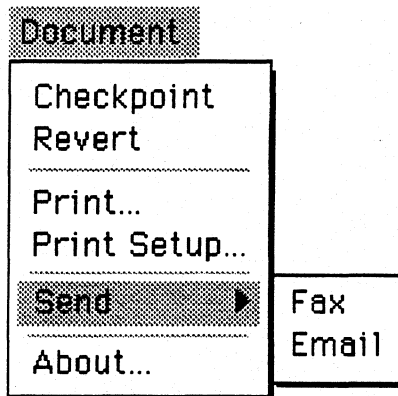


**Figure 33: Lists in Menus Don't Need Dividing Lines**

## **Hierarchical Menus**

The PenPoint toolkit supports hierarchical menus.

A right-pointing arrow to the right of a menu command tells the user that tapping that command will display a submenu, as shown in Figure 34.



**Figure 34: Submenu**

As a general rule, keep the menu hierarchy to one level. Hierarchies deeper than a single level are cumbersome to use. You can always avoid a deep hierarchy by either making the hierarchy broader at the top level or putting commands on other types of controls, such as option sheets or palette lines.

## **Issues**

We are considering adding some kind of facility to allow the user to convert modal, one-shot menus into modeless, floating menus for repeated use. How important is this feature?

## Chapter 6: Dialog Sheets and Option Sheets

This chapter gives guidelines for using dialog sheets and option sheets.

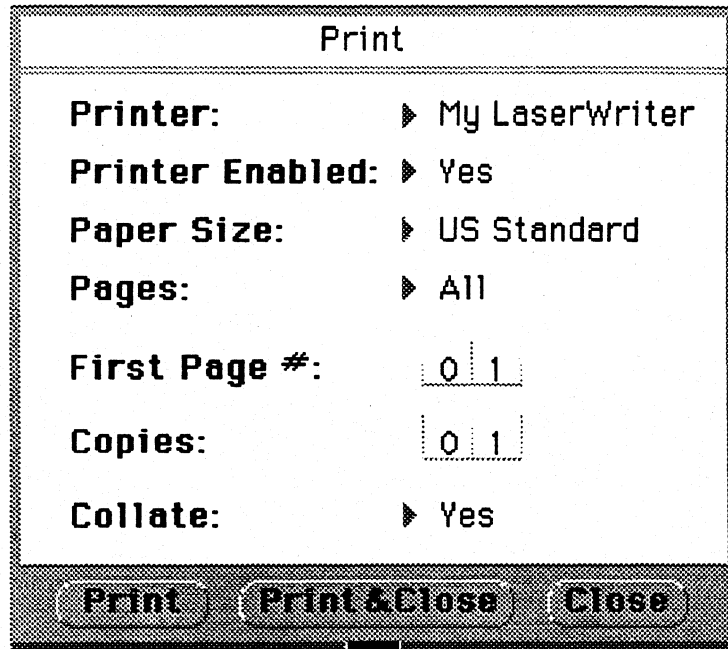
Topics covered include:

- *Dialog Sheets*. For allowing the user to set parameters to commands.
- *Option Sheets*. For allowing the user to set options for application objects. Includes conventions for tracking the selection, refreshing option sheets, clean and dirty controls, etc.
- *Layout of Controls*. Default two column layout, pop-up vs. inline lists.
- *Multiple Sheets*. Multiple sheets controlled by a pop-up list in the title line.
- *Modeless and Modal Sheets*. When to use modal sheets.

## **Dialog Sheets**

Use dialog sheets to allow the user to specify the parameters to commands.

Figure 35 shows a typical dialog sheet, for the **Print** command.



**Figure 35: Print Sheet**

## Option Sheets

You can think of option sheets as a specialized type of dialog sheet, in which the command is "Apply the settings on the sheet to the selected object."

Figure 36 shows a typical option sheet, for the PenPoint text component.

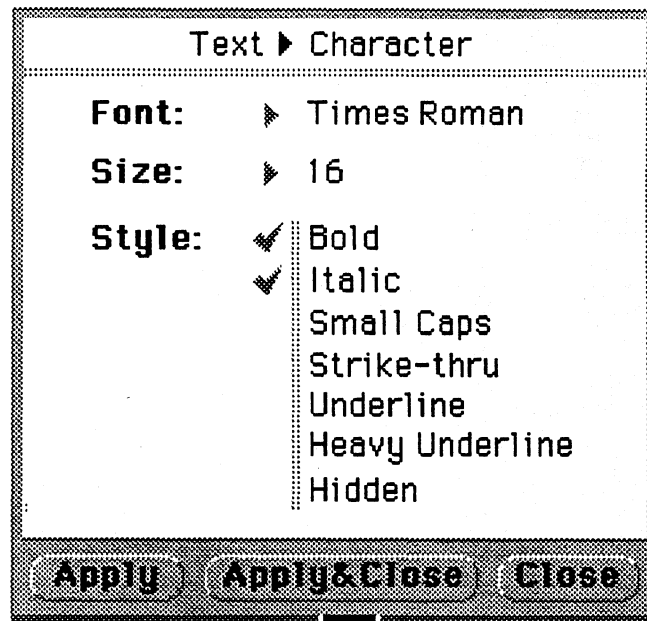


Figure 36: Text Option Sheet

### *Role of Option Sheets in PenPoint*

Option sheets are used widely throughout the PenPoint environment to allow the user to modify:

- application-specific objects such as text in a word processor. Typical text options include font, point size, paragraph justification, etc.
- individual documents. Standard document options include the title and author of the document, whether the document is editable and deletable, whether its menu line and cork margin are shown, etc.
- the overall PenPoint environment. Tapping on the Preferences icon brings up the PenPoint preferences sheets, allowing the user to set options for the environment as a whole, such as portrait or landscape layout, whether the bell sounds when warning or error messages are displayed, system font size, etc.

### *Option Sheets for Different Types of Objects*

If your application presents objects that appear to the user as clearly distinct types — text, figures and bitmaps are typical examples — it is usually best to present a separate option sheet for each type of object.

### *Invoking Option Sheets*

The user brings up an option sheet either from the a menu (the standard **Edit** menu, or an **Options** menu if present) or by making the checkmark gesture over the object of interest.

### *Dynamic Behavior of Modeless Option Sheets*

Option sheets are *modeless*, in that the rest of the screen remains responsive to input from the pen while the sheet is displayed.

The remainder of this section describes the dynamic behavior of option sheets — how they respond as the user makes a checkmark over an object or taps to change the selection.

It is very important that the conventions described below be implemented consistently whenever option sheets are used. In recognition of the importance of the option sheet behavior, the PenPoint toolkit includes a protocol to support consistent implementation by applications.

### *Response to Checkmark Gesture*

When the user draws the checkmark gesture over an object that is not selected, first select the object that was the target of the gesture, then display its option sheet. It is important to first select the object, so that the user can clearly tell which object is associated with the sheet.

This implies that you need to decide what the most natural and useful “default” object is in the context of your application. For example, in the Table of Contents the default object is the document, while in text the default object is the word (not the character) under the gesture.

If an option sheet is already up when the user draws a checkmark on another object (of the type appropriate to the option sheet) first select the object, then update the sheet's contents to reflect the current state of the object.

### *Relationship to the Selection*

Any time there is no current selection, or the current selection is not of the option sheet's type, dim the **Apply** and **Apply & Close** buttons, since there is no target for the actions.

### *Clean and Dirty Controls*

Option sheets have the concept of "clean" and "dirty" controls. Clean controls are shown in dark grey; dirty controls in black.

When the sheet initially comes up (or is updated by the drawing of a checkmark) all the controls are clean. When the user modifies a control, it is marked dirty.

When the user taps the **Apply** button, only the dirty controls are applied. This distinction allows the user to tell unambiguously which options will be affected by tapping **Apply**.

In most cases, this distinction is not important to the user. But there is one situation in which it is essential: when the selection contains multiple objects, that don't all share the same setting for the option.

Let's take a common example from text. Suppose an entire paragraph is selected. Most of the font is normal, but there are a few bold words and a few italic words. Since the setting would take its value from the first letter in the selection, neither bold nor italic would be checked. The user needs to know whether the entire paragraph will be made not bold and not italic when **Apply** is tapped. The clean/dirty convention satisfies this concern.

### *Copying Options from One Object to Another*

If the user selects another object while an option sheet is displayed, the toolkit automatically marks all the controls as dirty.

This allows the user to easily copy properties from one object to another by simply bringing up the option sheet for the source object, selecting the destination object, and tapping **Apply**.

## **Modeless and Modal Sheets**

By default, PenPoint dialog and option sheets are modeless.

You can also specify that a sheet be *modal*, in which case pen input to the rest of the screen is blocked as long as the sheet is displayed.

In most cases, it is better to use modeless sheets, because they are less restrictive to the user. For example, the user can bring up an option sheet, and then go through a document, repeatedly selecting and modifying different objects. The user can also interrupt the use of the option sheet to do something else, without needing to dismiss the sheet and then bring it up again.

Sometimes there is a good reason to restrict input to the sheet itself. For example, it may be necessary to control access to the application's data in order to maintain the integrity of the data.

Suppose a database application presented a dialog sheet to modify the result of a database query. It's quite possible that a modeless sheet would allow the user to change the state of the program's data such that it would be impossible or impractical to update the contents of the sheet. In such a case using a modal sheet can eliminate a whole class of problems.



## Command Buttons

The light grey area at the bottom of dialog and option sheets is called the *command line*, and the buttons found there are called *command buttons*.

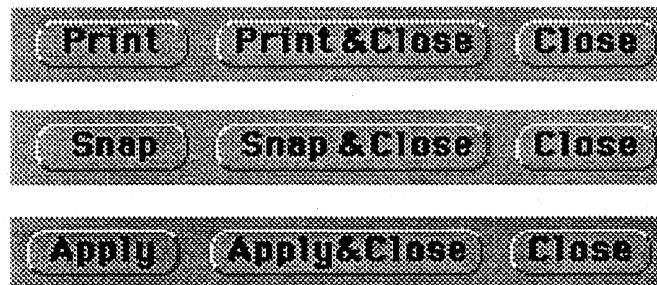
### Standard Modeless Buttons

In the simplest case the sheet has a single command — as in the case of the Print Sheet shown at the beginning of this chapter.

For such cases the rule is to use three buttons: one to execute the command and leave the sheet up, another to execute the command and close the sheet, and a third to close the sheet without taking any other action.

This button convention takes advantage of the modeless nature of the sheet by allowing the user to use the sheet one time or repeatedly.

For the sake of consistency, the three buttons should be named and ordered as in the examples shown in Figure 37.



**Figure 37: Standard Modeless Command Buttons**

### Standard Modal Buttons

For modal sheets that have a single command, the convention is to have just two buttons — one to invoke the command and dismiss the sheet, and another to dismiss the sheet without taking any action.

The latter button should always appear on the right, and be labelled **Cancel**. So, for example, if the Print Sheet were modal, its buttons would be labelled as shown in Figure 38.

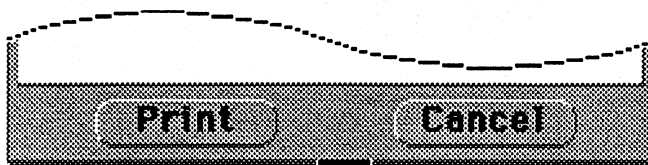


Figure 38: Standard Modal Command Buttons

The word **Cancel** is appropriate in two senses: the button cancels both the mode, and the the pending operation.

The **Close/Cancel** convention also serves the function of giving the user a consistent cue as to whether a given sheet is modal or modeless.

### Non-standard Buttons

Sometimes the standard conventions described in the previous sections aren't sufficient. The basic command for a dialog sheet may have variations, as in this example from the Find sheet.

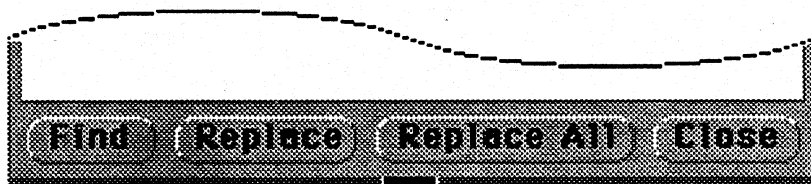


Figure 39: Command Buttons for Find Sheet

In such cases do the best you can to keep the buttons simple.

If possible, limit the buttons to a single row. If there are more buttons than will fit on a single row, consider grouping them into multiple sheets, as described in the later section on *Layout*.

### Command Buttons and Other Buttons

Only buttons that actually affect something outside of the sheet should appear on the command line. Any other buttons should be placed within the body of the sheet.

Take the example of a screen snapshot program. There is a need for a button that the user can tap to go into "drag mode" to specify the area to be captured in the snapshot — let's call it the **Aim** button. It might seem at first glance that this button should go on the command line, as shown in Figure 40.

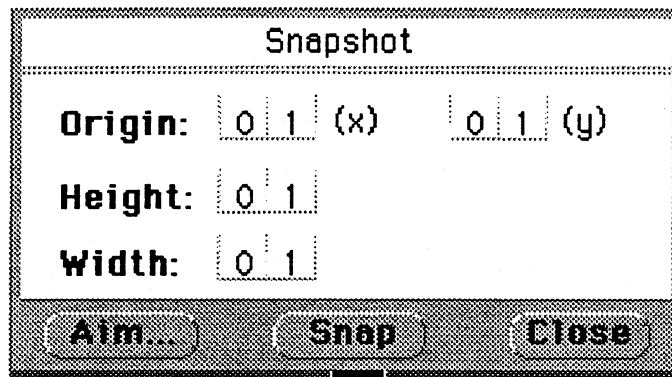


Figure 40: Non-Command Button on Command Line (Not Recommended)

This design has the weakness of losing the distinction between buttons (like **Snap**) that actually *effect* the command and those (like **Aim**) that only *set up* the command. Even if the user never consciously thinks in terms of this distinction, it is important to maintain it consistently, because it reflects the structure that underlies the interface.

Figure 41 shows a better solution.

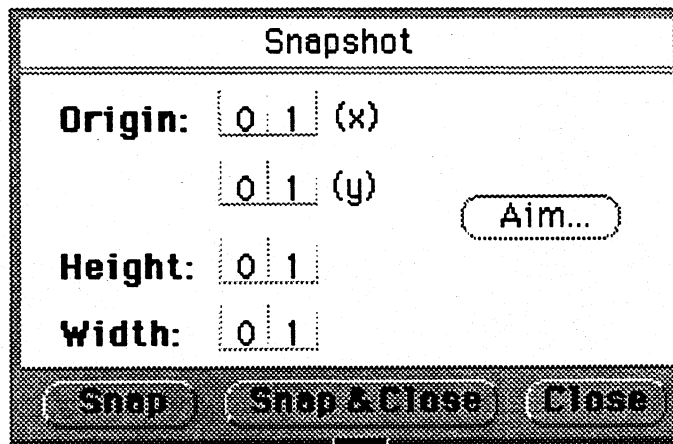


Figure 41: Proper Use of Command Line

The **Aim** button is in the body of the sheet, close to the parameters that it modifies. This also makes room in the command line for the standard three buttons.

## **Layout of Controls**

This section gives guidelines for laying out the controls in both dialog sheets and option sheets.

### *Standard Layout*

The default layout supported by the toolkit is designed to be simple and readable. It has two columns: the labels are on the left, the values are on the right.

Both columns are left-justified. The text in the value column line up vertically, whether the control is a choice list, a pop-up list, or a text field.

The labels are shown in bold, followed by colons.

All parts of the control — both label and value — are aligned on the same baseline.

### *Non-standard layouts*

If the default two-column layout does not suffice for your purposes, the PenPoint toolkit will support more complex layouts.

The most important guideline is to align all the controls along a clear grid. By following this basic rule of graphic design, you can maintain readability even in quite dense layouts. When information is not aligned on a grid, readability is degraded.

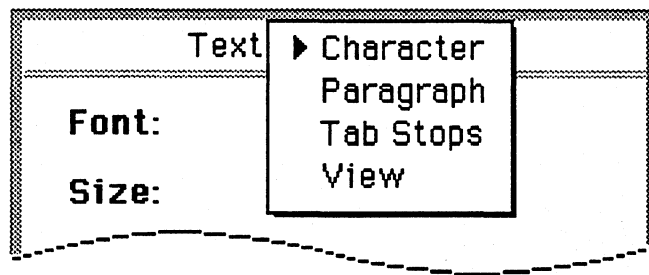
*Single vs. Multiple Sheets.*

As a general rule, it's a good idea to keep a given sheet simple and compact — six to ten controls to a sheet is a good limit. The larger the sheet, the more of the screen it will obscure when displayed. PenPoint sheets are modeless, so that the user can leave them displayed and use them repeatedly. This feature becomes less useful the larger the sheet becomes.

The basic technique to keep the sheets simple and compact is to organize the controls into separate categories, each with its own sheet.

When you create multiple sheets, the PenPoint toolkit automatically generates a pop-up list containing the titles of all the sheets, and places this control in the sheet's title line, following the overall title for the sheet.

Tapping on the arrow displays the menu with the names of all the sheets. Figure 42 shows the text option sheet with its menu open.



**Figure 42: Menu for Text Option Sheet**

The arrow in the title-line gives the user a consistent visual cue that there are multiple sheets.

Multiple sheets allow you to present complex functionality in a simple way by unifying several related commands. The user need only make the checkmark to bring up the most-recently-used sheet, then turn to any of the other sheets via the title-line menu.

Whenever possible, list each of the sheets from an **Options** menu, as recommended in the later chapter on *Putting the Building Blocks Together*. This gives the user a choice of two complementary paths to the desired command: making a checkmark to bring up the most-recently-used sheet, or using the menu to go directly to any sheet.

### Pop-up vs. Inline Choices

As described in the chapter on *Controls*, PenPoint provides two forms of choice lists: an inline form in which all the choices are shown, and a more compact pop-up form in which only the current choice is shown.

Using only one form of list on a single sheet has the advantage of ensuring both visual clarity and consistent behavior for choosing from the lists. First consider using the inline form, which is easier to browse and use, for all of the lists on the sheet. If that would make the sheet too tall, use the pop-up form.

This is not a hard and fast rule. There are two good reasons to mix inline and pop-up lists on a single sheet:

- If the controls fall clearly into frequently-used and seldom-used controls, consider presenting the frequently-used lists inline and the seldom-used lists as pop-ups.
- If there is a natural grouping of controls, consider showing the related controls in the same form.

The document *Access* sheet, shown in Figure 43, provides a good example of both these guidelines.

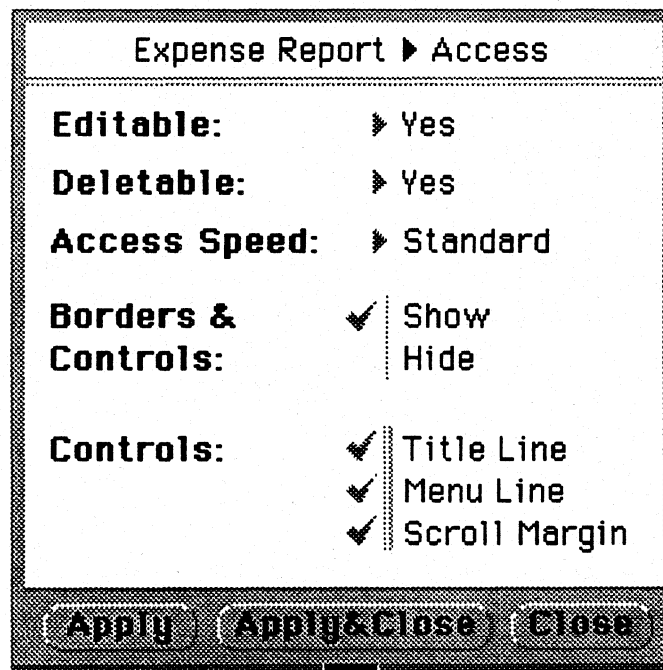


Figure 43: Document Access Sheet

The **Borders & Controls** toggle would ordinarily be shown as a pop-up list, like the first three lists on the sheet. But since its function is so closely related to the multiple checklist below it, and it is the most frequently-used control on the sheet, it is shown inline, to reinforce the relationship.

### *Fill-in vs. Overwrite Fields*

As described in the chapter on *Controls*, PenPoint provides two forms for text fields: underlined fill-in fields and overwrite fields that are segmented into discrete character boxes.

Each of these forms has its advantage. The overwrite fields are easier to use, since the user can edit them directly without first tapping to pop up an edit pad.

The disadvantage of the overwrite fields is that they are usually much larger than fill-in fields. The sizes of these two types of field are controlled by different user preferences: fill-in fields are shown in the size of the system font, while overwrite fields take their size from the **Box Size** preference on the Handwriting Preference Sheet. By default, the overwrite fields are larger, to make it easy to write accurately in them.

By default, PenPoint sheets use overwrite fields for numeric fields, and fill-in fields for text fields, which are usually longer.

Given tradeoff between editing efficiency and compactness, it's up to you to decide which form to use. Unless there's a compelling reason to mix them, use only one form on a single sheet.

### *De-activating vs. Hiding Controls*

As with menus, when a control is not available for any reason, you should let the user know by de-activating it.

Often a control is active or inactive depending on the state of the control above it. Figure 43 on the previous page contains such a pair. The list labelled **Controls** is only active when the switch labelled **Borders & Controls** is set to **Show**. In the case of such related pairs, it is best to activate and de-activate the dependent control immediately as the dominant control is changed, rather than only after the **Apply** button is tapped. This helps to make the semantic relationship between the two controls clear to the user.

It is also acceptable to hide the dependent control completely when it is irrelevant. If you do so, put the control that comes and goes at the bottom of the sheet if possible, so that it doesn't leave a gap in the middle of the sheet.

With de-activating and hiding, as with other stylistic choices, it is usually best to choose one method or the other and use it throughout your application.

## **Issues**

We are considering asking the user for confirmation when the user has pending changes and turns to another sheet on a multiple option sheet. Is this a good idea?

We are considering changing the label of the **Close** button to read **Cancel** when there are pending (unapplied) changes. Is this a good idea?



## Chapter 7: Status, Warning and Error Feedback

This chapter gives guidelines for giving feedback to the user, including:

- Busy clock
- Wording guidelines for messages and button labels
- Progress and completion messages
- Confirmation notes
- Error notes
- Timing-triggered notes
- Audible feedback for warning and errors
- Message lines

## Busy Clock

It is very important that the user have some indication that the machine is alive if the result of a command doesn't appear instantly.

PenPoint provides a standard *busy clock* for this purpose. It is very important to use the busy clock consistently, because your user will be expecting it, since it is used throughout the PenPoint environment.

To make it more visible to the user, the busy clock is animated, with one hand rotating through eight positions, as shown in Figure 44.

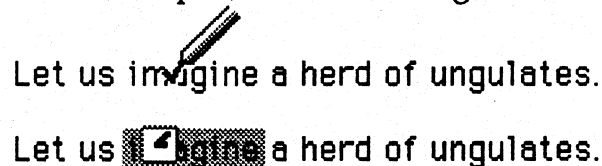


**Figure 44: Frames of Animated Busy Clock**

### *Busy Clock Location*

In traditional mouse-based interfaces, the mouse pointer, which is always visible on the screen, is used as the busy indicator. The situation is a little different with a pen-based interface. Because there is usually no pointer on the screen to provide a visual focal point, it is especially important to position the busy clock so that it will be clearly visible to the user.

In most cases, it is best to display the busy clock at or near the point that the user invoked the operation with the pen, as shown in Figure 45.



**Figure 45: Busy Clock at Location of Gesture**

In the above example the user has drawn a checkmark gesture to bring up the option sheet for a word. The busy clock is centered on the pen-down point of the gesture.

It is often helpful to position the busy clock precisely in relation to the object of interest. For example, when the user taps on an icon, the busy clock appears right over the icon, as shown in Figure 46.

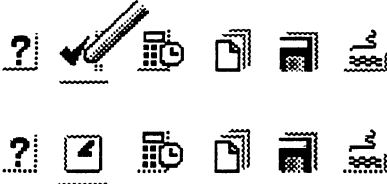


Figure 46: Busy Clock On Top of Icon

When the operation is invoked from a button, center the busy clock just above the button, as shown in Figure 47.

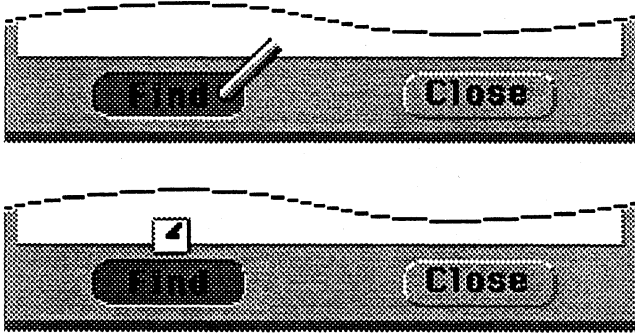


Figure 47: Busy Clock Above Button

This eliminates the visual interference caused if the clock is centered on the pen-down point of the tap, partially overlapping the icon or button.

In some cases it works best to place the clock not at the point of the gesture, but in a consistent location with reference to the surrounding sheet. For example, when the user turns from one option sheet to another, PenPoint places the busy clock at the far right of the sheet's title line, as shown in figure 48.

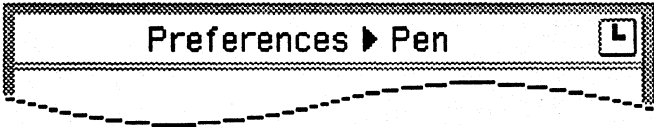


Figure 48: Busy Clock At Right of Title Line

The clock is still close enough to be in the user's field of vision, and is more easily discerned than if it were overlapping the characters in the title.

## **Wording Guidelines**

This section gives guidelines for the wording of the messages and button labels in notes.

### *Wording of Messages*

- Use normal, mixed case for the message — don't use all upper case letters for emphasis.
- Don't use exclamation points — they give an exaggerated, cartoon-like impression. And they're not really necessary!
- Never use words that may have offensive connotations, such as “abort.”
- Put the message in the user's terms, not the programmer's terms. For example, “Not enough memory for operation — press the restart button, then try again” is more helpful than “Heap allocator: unable to malloc 4 K block.”

### *Wording of Button Labels*

PenPoint supports modal notes for the display of progress, completion, confirmation and error messages, as described later in this chapter.

In labelling buttons on modal notes, follow the convention described in the preceding chapter for modal dialog sheets: the name of the command, and **Cancel**. For modeless notes, use **Close** instead of **Cancel**.

This convention of using active verbs makes the buttons unambiguous. It also allows the buttons to stand on their own — the user can interpret them at a glance, without needing to read the message carefully.

Don't label buttons **Yes** and **No**. That forces the user to read the labels carefully to know how to interpret them — does **Yes** mean “Yes, proceed with the operation” or “Yes, I want to cancel the operation?”

## **Progress and Completion Messages**

If an operation is very lengthy — or especially important to the user for any reason — it's a good idea to supplement the busy clock with an explicit message while the operation is in progress.

You can use either a pop-up note or an in-line message area to display progress messages, as described on the following two pages.

*Pop-up Progress/Completion Note*

The note shown in Figure 49 tracks the progress of a disk formatting operation.

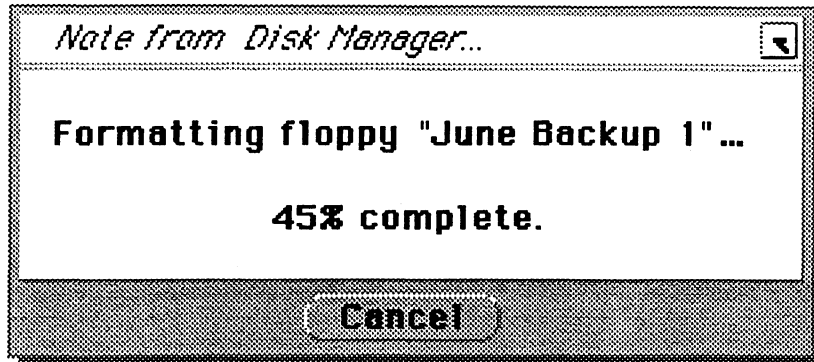


Figure 49: Progress Note

Note the busy clock at the right of the note's title line.

Whenever possible, you should put a **Cancel** button on such progress notes to allow the user to cancel long operations.

There are two ways to handle the normal completion of an operation. The simplest is to take down the note programmatically when the operation is done.

If it is important that the user acknowledge the completion of the operation, then leave the note up, change the message to reflect the completion of the operation, and change the label on the button from **Cancel** to either **OK** or **Continue**, as shown in Figure 50.

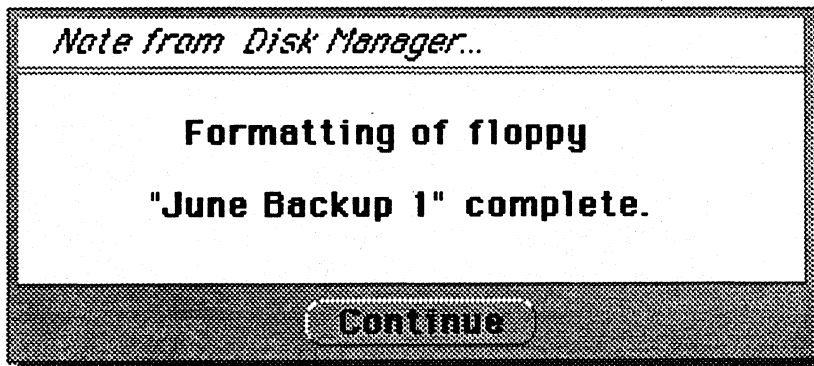
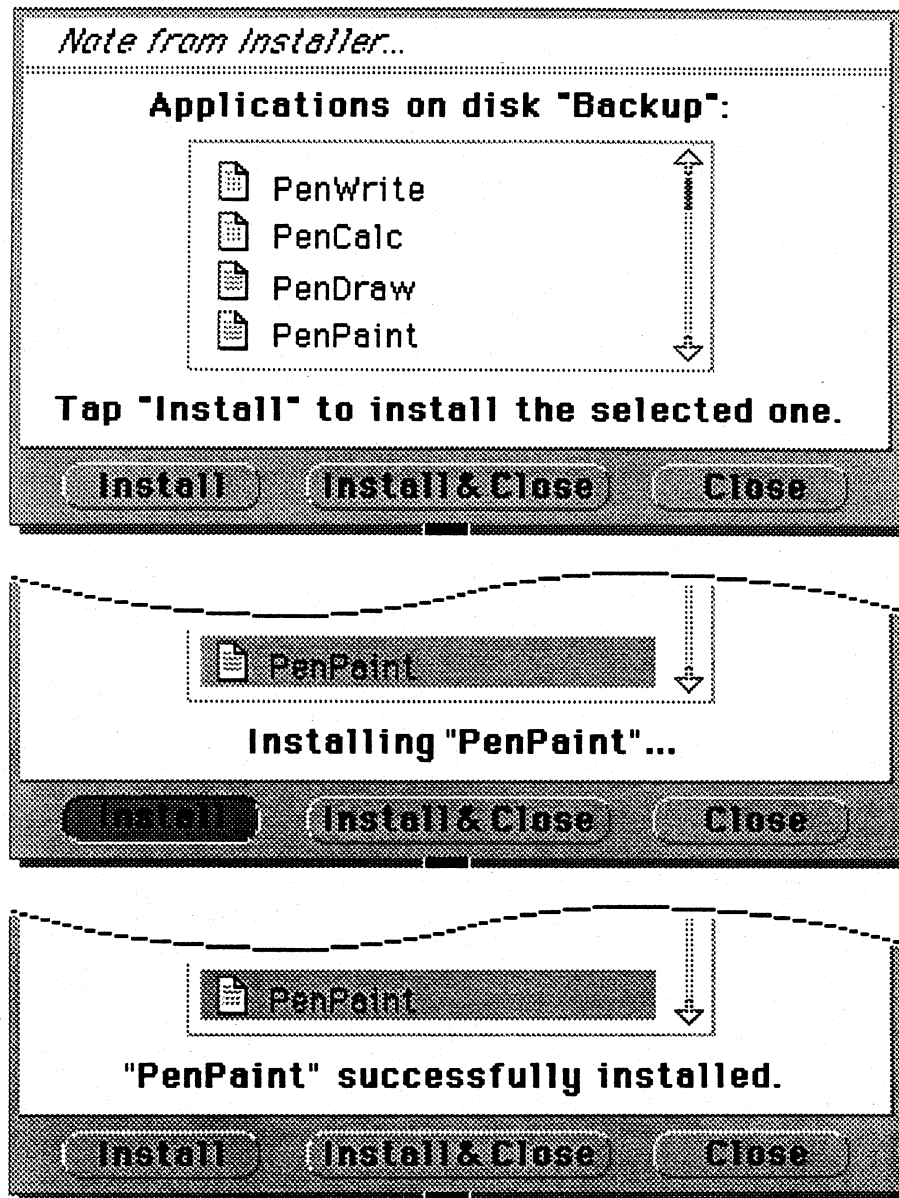


Figure 50: Note Marking Successful Completion of Operation

*In-Line Progress/Completion Message*

When possible, put a progress message in a note that is already up. This disturbs the screen less.

Figure 51 shows a progress message embedded in a pre-existing note.



**Figure 51: Progress and Completion Message as Part of Note**

## Confirmation Notes

Before proceeding with operations that are irreversible and that destroy data, you should protect the user by asking for confirmation.

Figure 52 shows a typical confirmation note.

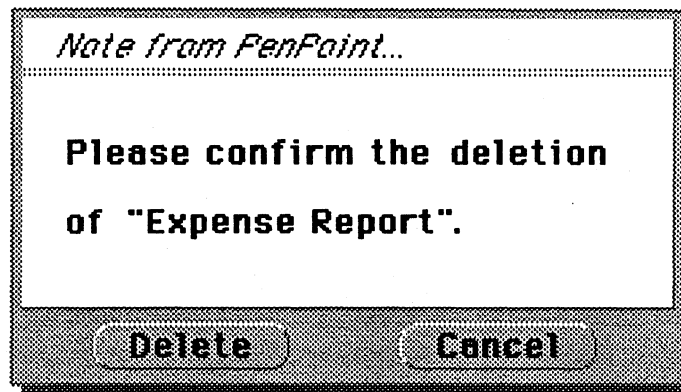
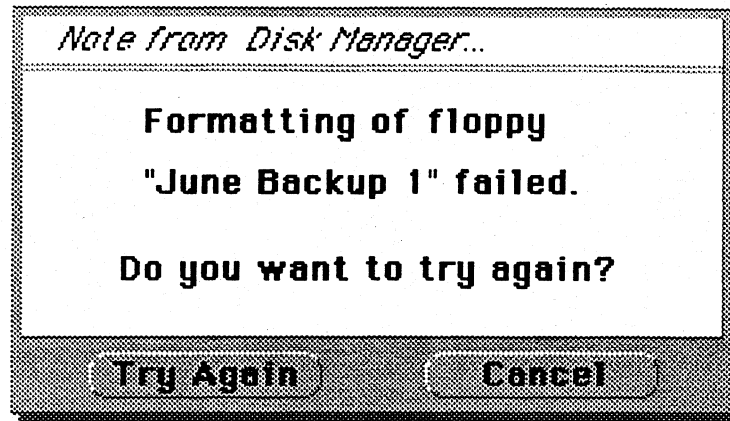


Figure 52: Confirmation Note

As with all notes, use active verbs for button labels, rather than Yes and No.

## **Error Notes**

Figure 53 shows a typical pop-up error note.



**Figure 53: Error Note**

Whenever possible, tell the user what he or she can do next. In the above example, the options are to try the operation again or to cancel the operation.

If you can give the user a short hint as to what they can do to work around the problem, by all means do it.

As with all notes, use active verbs for button labels, rather than **Yes** and **No**.



## Timing-Triggered Notes

In some situations it may be appropriate to display a note for a few seconds, then automatically dismiss it and continue. You can use this technique both for completion notes and error notes.

For example, when the user tries to turn to a notebook page, and the application framework is unable to open the document on that page, it displays the note shown in Figure 54.

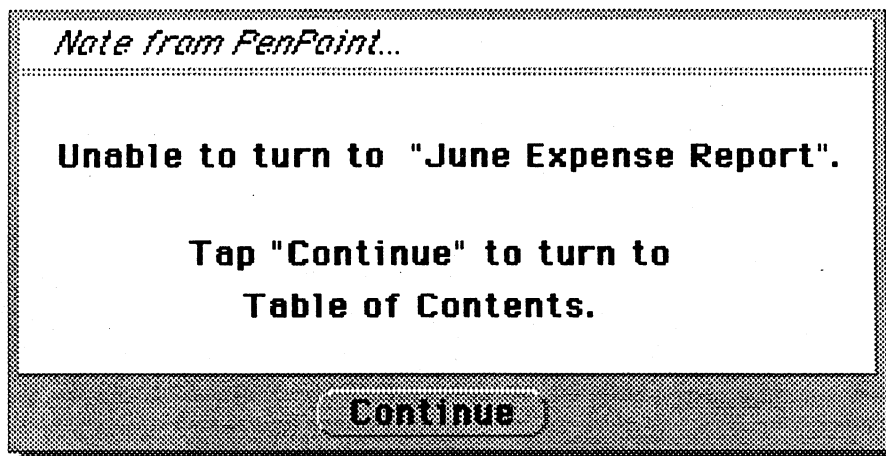


Figure 54: Timing Triggered Error Note

If the user doesn't tap **Continue** within 5 seconds, the application framework takes down the note and turns back to the Table of Contents.

Use timing-triggered notes with caution. Never use them in situations where it is really important that the user see the note.

Another problem with timing-triggered notes is how long a timeout interval to use. Dismiss the note too quickly and the user will strain to see it, or miss it entirely. Leave it up too long and the user may become impatient waiting for it to go away. Given that different user will vary in how fast they read, it's not clear that one interval will work for the range of users that will use your application.

If you do use timing-triggered notes, it is especially important to test the interval on users.

## **Audible Feedback for Warning and Errors**

PenPoint is designed to be used anywhere — in meetings, during interviews, in classrooms, etc. — without intruding into the social situation.

For this reason PenPoint provides a user preference for audible feedback, which by default is disabled.

You can use audible feedback for minor errors, subject to these two guidelines:

- Always check the user preference before providing audible feedback. Never provide audible feedback when the user has indicated that it is not desired.
- Never rely solely on audible feedback, since it might be turned off by the user. Always accompany the audible feedback with some kind of visual feedback (such as briefly flashing the title line.)

If the user has enabled the audible feedback preference, notes will be accompanied by audible feedback when they are displayed. You can override this behavior, so that a particular note is never accompanied by audible feedback, regardless of the preference setting.

## Message Lines

If you need to give many messages or instructions to the user, you can do so by providing a message line as part of the layout of your application.

55

Figure 59 shows an application with a palette line for different drawing modes, and a message line giving instructions for each mode.

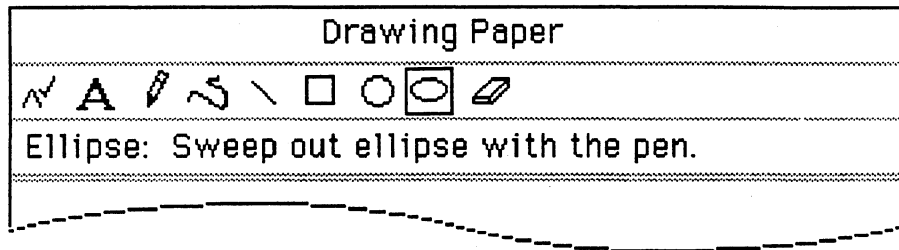


Figure 55: Message Line

Note that the message line is on the top rather than the bottom. Because of the way the PenPoint computer is held in the hands, the top half of the screen is more easily seen than the lower half, which is more often obscured by the user's hands.

If you do put a message line at the bottom, make sure that you do adequate user testing to verify that users see the messages.

One of the considerations underlying the use of a message line is the basic issue of how much feedback to give to the user. Too little and the program may be seen as terse and unfriendly. Too much feedback will get in the user's way, and may be seen as cumbersome or even condescending. You need to know your audience.

## Issues



## Chapter 8: Putting the Building Blocks Together

The preceding chapters have introduced the various building blocks provided by PenPoint — controls, menus, dialog and option sheets, etc.

With these building blocks in hand, we can step back and ask how they should be put together. Should a given function go on a menu, or an option sheet, a palette, or all three? Should all functions be put on menus? On option sheets?

This chapter gives principles and examples relevant to answering the basic question of how to present your application's functionality.

Topics include:

- Dual command path — controls and gestures.
- Layering of functionality to hide complexity.
- Allowing the user to configure the interface to your application.
- When to use menus, palettes, and option sheets.
- When to depart from the standard building blocks.

## **Basic Guidelines**

This section mentions some basic guidelines for to keep in mind when thinking about how to present your application's functionality.

### *Dual Command Path — Controls and Gestures*

This basic PenPoint principle, introduced in Chapter 3, bears mention again here. To summarize the gesture guidelines:

- Throughout your design and development process, ask yourself how you can give your users the choice of using either visible controls — buttons, checklists, menus, palettes — or gestures.
- Use the core gestures whenever they are relevant to your application. Always follow the standard PenPoint usage for the core gestures.
- Use the non-core gestures and the capital letters as accelerators for frequently-used operations. Use gesture accelerators judiciously — the point isn't to find a gesture accelerator for every command, but to use them where they really make the user's job easier.

### *Layering*

This principle applies in designing any application, particularly those that present a rich set of functionality and will be used by a diverse set of people.

Don't overwhelm the user by presenting all of the commands at the same level. Layer the interface to hide complexity. Present the most important commands prominently, and put the rest behind the surface, on menus or option sheets that the user must open to see.

*User Configurability*

Think of how you can help the user by allowing him or her to configure the face presented by your application.

There are many ways to do this — by providing “novice” and “expert” modes, by letting the user put favorite commands on customized palettes, etc.

One degree of flexibility you should always provide is to allow the user to hide any non-essential palette line, message line or other region, to make more room for the display of data.

Add the controls for showing and hiding the non-essential regions to the standard document Access sheet, as shown in Figure 56.

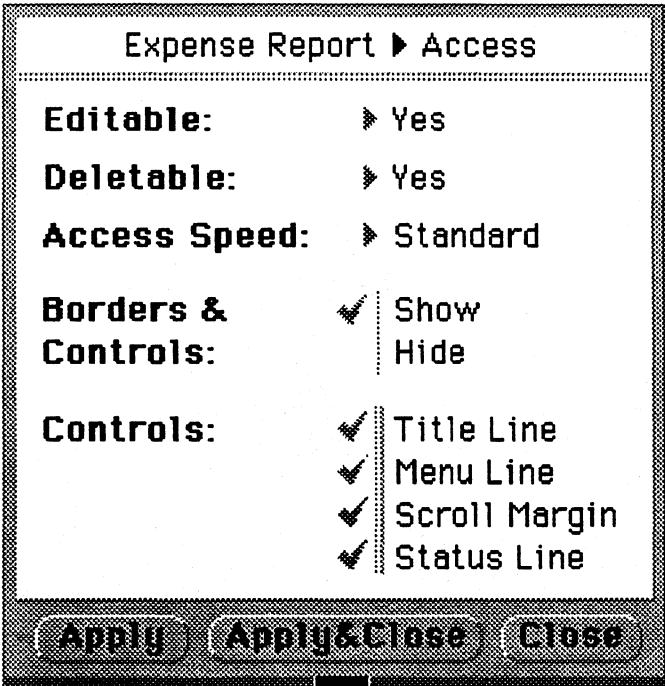


Figure 56: Status Line Control on Access Sheet

In this example the application has added a **Status Line** choice to the **Controls** list.

## Where to Put It: Menu, Option Sheet, or Palette?

Now we come to an issue that is very basic, and yet difficult to give hard and fast guidelines for — how to divide your application's functionality between menus, option sheets, and visible control areas such as palette lines and custom control regions.

No single user interface mechanism is “best” — each has its strength and corresponding weakness.

As a reference point, the table below summarizes the main strengths and weaknesses of menus, option sheets, and palettes.

<u>Mechanism</u>	<u>Strengths</u>	<u>Weaknesses</u>
Menus	<p>Menus invite the user to explore by tapping.</p> <p>Menus are easy and fast to invoke for a single operation.</p>	<p>Menus are inefficient for often used commands, or for commands that are typically issued in groups, since they disappear automatically as soon as one command is issued.</p>
Option Sheets	<p>Option sheets don't require the user to access each control by name. All the user need remember is to make the checkmark gesture over the object of interest.</p> <p>Because they are modeless, option sheets allow the user to easily change several settings at once, or to keep the sheet up and use it repeatedly.</p>	<p>Option sheets take longer to display than menus, and usually take up more screen real estate than menus (although the user can resize the sheet to show only the desired control).</p> <p>While option sheets are displayed, they obscure part of the screen, often requiring the user to drag them out of the way.</p>
Palettes	<p>Palettes provide the most visual invitation, and the most efficient access, since all the controls are visible at once.</p> <p>Palettes are typically compact, since they use small glyphs instead of longer textual labels.</p> <p>Palettes don't obscure any of the document's work area, and never need to be dragged out of the way.</p>	<p>Not appropriate for commands that can't be clearly expressed as a small glyph.</p> <p>Not appropriate for less frequently-used commands.</p> <p>Take up screen real-estate permanently.</p>

On the following page are two useful questions to ask when considering how to present your application's commands and options to the user.



1. *What are the most important, obvious objects that the user will see and deal with?*

In general, you should provide an option sheet for each type of object in your application, unless there's a compelling reason not to.

Whatever the object is — a span of text, a figure in a graphics editor, a cell in a spreadsheet, an appointment entered into a calendar, — the user should be able to view and modify the object's options by drawing a checkmark over it.

For example, suppose your application presents a list of sales contacts. Each contact in the list has associated information that the user can modify. You could call the command by a specialized name such as **Edit Entry** or **Update Contact Information**.

But that approach forces the user to learn and remember the specialized name you have chosen. By presenting the same command under the generic umbrella of **Options**, you lessen the learning curve. The user comes to your application already knowing about option sheets and how to invoke them. There's no need to even think in terms of a new command. The user simply makes a checkmark on the entry in the list, and already he or she is successfully using the application.

2. *Is there a set of commands or modes that are fundamental to your application, that the user will be using very frequently?*

If the answer is "yes," it is probably best to present the commands in a palette that is always visible, so the user never has to worry about popping up a menu or option sheet, or dragging an option sheet out of the way.

The above two guidelines will suggest what you should put on option sheets and palettes — now what about menus?

When designing your menu line, there are two general approaches you can take.

One is to put all the important functionality on the menu line, so that the menu line, as the primary way of issuing commands, becomes indispensable.

The other is to put less functionality on the menu line and more on other input paths — gestures, palette lines, or option sheets. That way the user can turn off the menu line and continue using the application. In the notebook Table of Contents, for example, the user can turn off the menu line and browse through the notebook contents, turn to documents, turn tabs on and off for specific documents, move, copy or export documents, etc.

## **When to Depart from the Standard Building Blocks**

Don't depart gratuitously. There's not much added value in being different for differences sake, and there may be a cost, if it requires the user to learn new variations. For example scroll margin visuals.

But of course there are many situations in which it is appropriate to depart. Guidelines and standards are not meant be rigidly applied, to produce a monotonous scene. In the oft-quoted words of Ralph Waldo Emerson, "A foolish consistency is the hobgoblin of small minds."

The best reason to depart from the standard visuals is to emulate a real world object, such as a calculator, control panel, rolodex, telephone message slip, etc. In such cases by all means be realistic.

In fact, being "realistic" has all the benefits cited for consistency in interfaces — reducing learning time by taking advantage of the user's familiarity from other contexts. Only with realistic objects the consistency is with respect to things from other domains that the user is familiar with, instead of the domain of the standard PenPoint user interface.

## **Issues**

## **Section III: Standard User Interface Elements**

PenPoint, through the Application Framework, provides support for the consistent presentation of many operations that in traditional operating systems are left to each application to handle in an ad hoc way.

This section describes the standard user interface elements, including:

- Standard menus and commands
- Standard application option sheets
- Icons
- Help



## Chapter 9: Standard Menus and Commands

This chapter describes the standard menus and commands supported by the Application Framework.

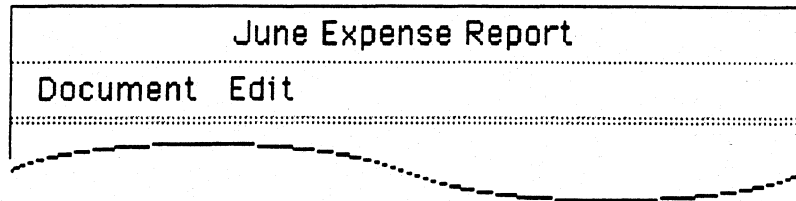
Topics covered include:

- Standard menus — **Document and Edit**
- Recommended menus — **View and Options**
- Standard commands, including **Checkpoint and Revert, Print, Move, Copy and Delete, Find and Spell, Import and Export.**

## Standard Menus

The PenPoint Application Framework provides support for two standard menus, **Document** and **Edit**.

Figure 57 shows the default menu line with the standard menus.



**Figure 57: Default Menu Line with Standard Menus**

These menus represent two basic categories of functions that are widely applicable across applications:

- **Document** is for commands related to the entire document (e.g. **Checkpoint** and **Revert**) and to the outside world (e.g. **Print** and **Send**).
- **Edit** is for commands related to editing the objects in the document (e.g. **Move**, **Copy**, and **Delete**).

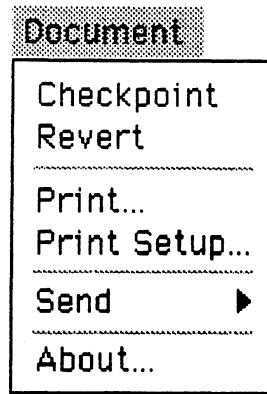
Follow these guidelines for using the standard menus:

- If the functionality covered by these menus is relevant to your application, present it in these standard menus, unless there is a compelling reason to do otherwise.
- If you do use the standard menus, they should always be the first and second menus on the menu line.
- If any of the default commands on the standard menus are not appropriate for your application, remove the command entirely (rather than deactivating it.)

The sections that follow describe both standard menus.

*Document Menu*

Figure 58 shows the default **Document** menu.



**Figure 58: Default Document Menu**

The commands are:

- **Checkpoint.** Files the document in its current state.
- **Revert.** Restores the document to the state it was in at the last checkpoint.
- **Print.** Displays the Print Sheet for printing the document.
- **Print Setup.** Displays the Print Setup Sheet for setting print-related options such as headers and footers, margins, etc.
- **Send.** Displays a submenu with commands to invoke each of the currently installed services, such as fax, email, etc.
- **About.** Displays the application option sheets. The Application Framework provides three standard sheets: **Title & Info**, **Access** and **Application**. See the chapter on *Standard Option Sheets* for details.

*Customizing the Document Menu*

The Application Framework implements the **Checkpoint** and **Revert** commands for all documents that follow the normal PenPoint model and keep a second copy of their data in the file system. In fact, the Application Framework automatically checkpoints such documents whenever the user turns away from the document's page.

You may choose to keep only one copy of your application's data. In that case you need to either implement the **Checkpoint** and **Revert** commands yourself or remove them from the menu.

Edit Menu

Figure 59 shows the default Edit menu.

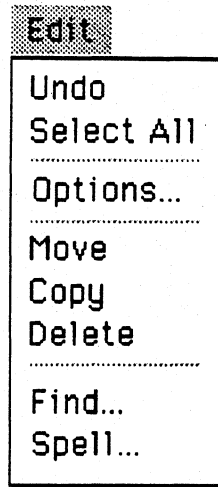


Figure 59: Default Edit Menu

The commands are:

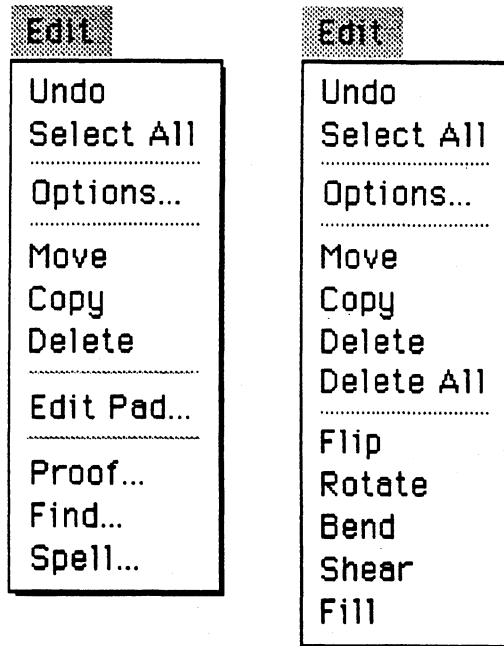
- **Undo.** Reverses the effect of the most-recently-executed operation. At a minimum, all applications should support a single-level undo. When possible, modify the label of the command to indicate which operation would be undone (e.g. **Undo Delete**, or **Undo Move**.)
- **Select All.** Selects the entire contents of the document.
- **Options.** Brings up the option sheet for the currently-selected object in the application.
- **Move.** Puts the selected object into **Move** mode (see the chapter on *Move and Copy* for details).
- **Copy.** Puts the selected object into **Copy** mode (see the chapter on *Move and Copy* for details).
- **Delete.** Deletes the selected object.
- **Find.** Displays the **Find** sheet, with the starting point for the search set to the beginning of the document.
- **Spell.** Displays the **Spell** sheet, with the starting point for the spelling check set to the beginning of the document.



*Customizing the Edit Menu*

You will probably want to tailor the Edit menu to fit your application.

Figure 60 shows two typical Edit menus.



**Figure 60: Customized Edit Menus**

On the left is the menu for PenPoint's text component. The **Edit Pad** and **Proof** commands have been added.

The menu on the right is for a drawing program. Note that:

- The **Find** and **Spell** commands, which don't apply to drawings, have been removed.
- A set of commands to act on figures have been added at the bottom of the menu.
- A **Delete All** command has been added immediately below the standard **Delete** command. This is the recommended wording and location for the command, rather than calling it **Erase** or **Clear** and putting it on the **Document** menu.

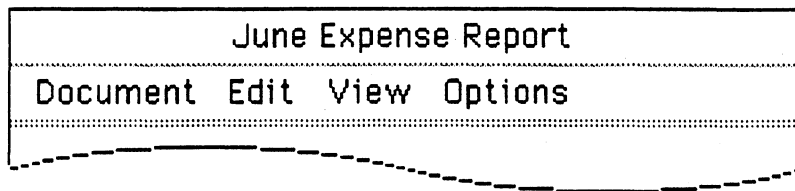
Follow this guideline for commands that create new objects:

- If your application has a single command for the creation of new objects, put it in the **Edit** menu. An example of this usage is the PenPoint Disk Manager, which has a **Create Directory** command.
- If you have separate commands to create objects of different types, it usually makes more sense to present them on a separate menu. So, for example, the notebook Table of Contents has a **Create** menu for creating new documents and sections, and the text component has an **Insert** menu for inserting new text and various other objects.

## Recommended Menus

You should consider using two additional menus, **View** and **Options**. These menus represent categories of functions that (while not implemented by the Application Framework as standard menus on the default menu line) are also widely applicable across applications.

Figure 61 shows the default menu line with the standard and recommended menus.



**Figure 61: Default and Recommended Menu**

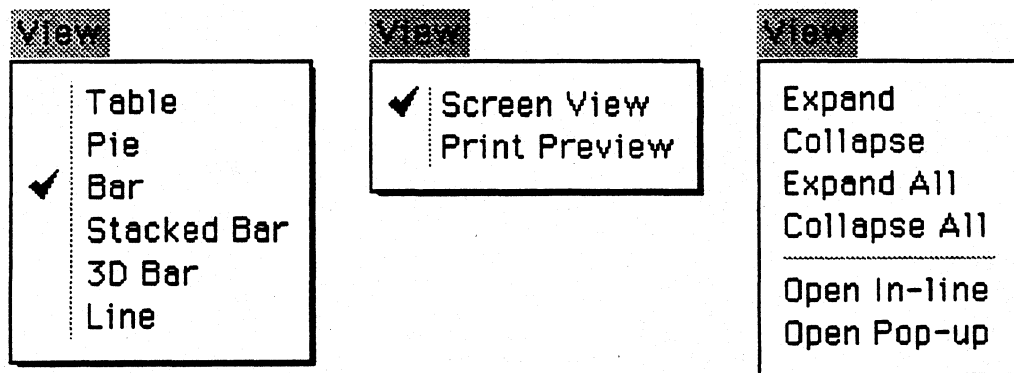
### *View Menu*

Whatever your application is, you will probably have commands to allow the user to modify how the application's data is presented.

As discussed in the chapter on *Using the Building Blocks*, you can present this information in a variety of ways — through menus, option sheets, or palette lines.

If you use do a menu, consider calling it the **View** menu, and putting it immediately to the right of the **Edit** menu.

Figure 62 shows examples of typical **View** menus.



**Figure 62: View Menus**

The menu on the left is for a charting application, the middle menu is for a word processor, and the menu on the right is for an outliner, in which categories can be expanded and collapsed, and entries can be opened.

### Options Menu

Many option sheets are actually multiple sheets, each displaying a different category of options, grouped together. For example, the option sheet for PenPoint's text component is comprised of separate sheets labelled **Character**, **Paragraph**, **Tab Stops**, and **View**.

In such cases you should remove the **Options** command from the **Edit** menu, and provide a separate **Options** menu listing each sheet.

This implements the PenPoint principle of giving the user a dual path — visual controls and gestures — for commands. The **Options** menu complements the checkmark gesture by allowing the user both to find out which sheets are available by browsing the menu, and to go directly to any sheet from the menu.

Figure 63 shows two examples of **Options** menus.

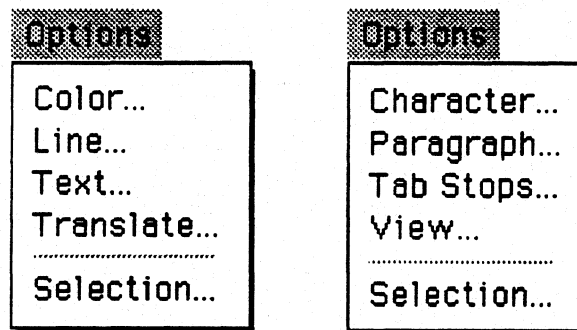


Figure 63: Options Menus

The menu on the left is for a drawing program; the menu on the right is for a text editor.

As shown in the above example, the command originally labelled **Options** in the default **Edit** menu should be called **Selection** in the **Options** menu. This command is still necessary, to allow the user to bring up the option sheet for embedded objects.

### Issues

Do we have the right set of commands on the standard menus?

Do we have the right recommended menus?

Some people have suggested a **Find Next** command on the **Edit** menu.

## Chapter 10: Standard Option Sheets

The Application Framework provides three standard application option sheets for information and options that are common to most documents.

Topics covered include:

- Accessing the application option sheets.
- The **Title & Info** sheet — for basic information about the document.
- The **Access** sheet — for controlling access to the document (e.g. whether can be edited or deleted, and whether standard document elements such as the menu line and cork margin are displayed).
- The **Application** sheet — for displaying information about your application and company.
- Customizing the standard option sheets.
- Adding application-specific option sheets.

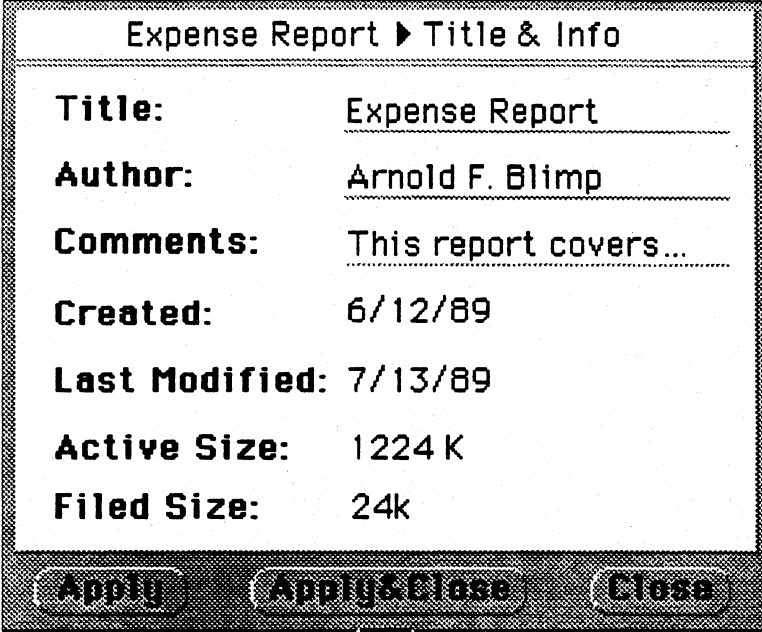
## Accessing The Application Option Sheets

The user can access these sheets either from the **About** command on the **Document** menu, or by making the checkmark gesture on the title line of the document or tool.

### Title & Info Sheet

The **Title & Info** sheet presents standard information about the document, including its title and author, a comments field, and its size and date of creation.

Figure 64 shows the default **Title & Info** sheet.



The screenshot shows a dialog box titled "Expense Report ▶ Title & Info". It contains the following information:

<b>Title:</b>	Expense Report
<b>Author:</b>	Arnold F. Blimp
<b>Comments:</b>	This report covers...
<b>Created:</b>	6/12/89
<b>Last Modified:</b>	7/13/89
<b>Active Size:</b>	1224 K
<b>Filed Size:</b>	24k

At the bottom of the dialog box are three buttons: **Apply**, **Apply & Close**, and **Close**.

**Figure 64: Title & Info Sheet**

## Access Sheet

The Access sheet contains options related to accessing the document, including controls to disable editing and deletion of the document, and controls to show and hide various standard components such as the menu line, cork margin, and borders.

Figure 65 shows the default Access sheet.

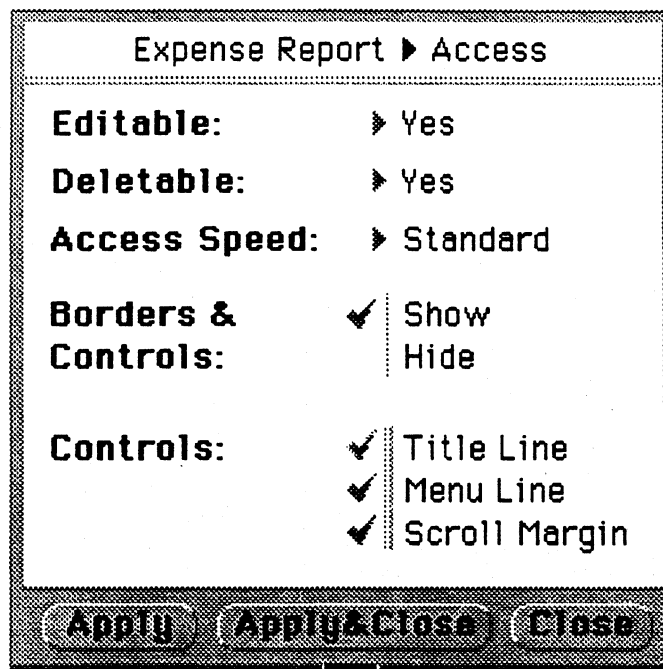


Figure 65: Access Sheet

As discussed in the chapter on *Putting the Building Blocks Together*, if you include a palette line, message line, or other control area or as part of the layout of your application, be sure to add it to the list of controls that the user can turn and off from the Access sheet.

## Application Sheet

The **Application** sheet contains information about the application controlling the document.

Figure 66 shows the default **Application** sheet.

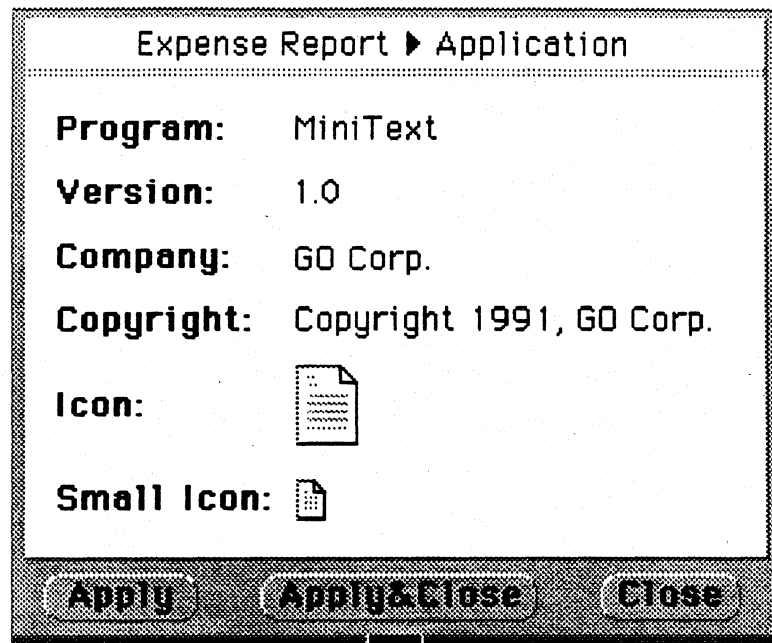


Figure 66: Application Sheet

The default layout shown above is provided to indicate the type of information that should go on the sheet. By all means provide your own customized sheet, with your company logo, fancy graphics, etc.

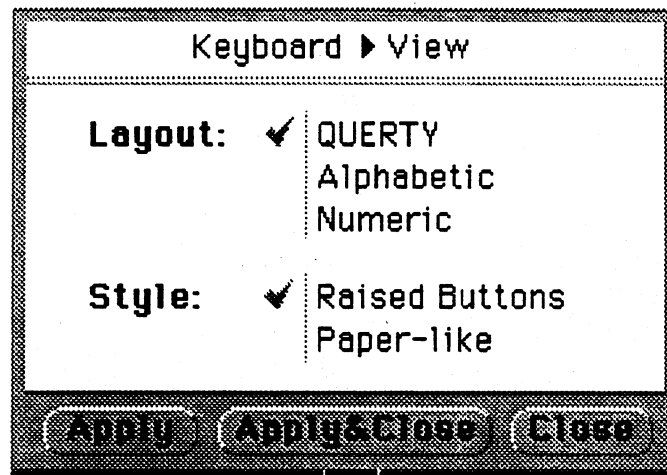


## Adding Your Own Application Option Sheets

The application option sheets are the place to put controls that apply to the application as a whole, as opposed to the objects within the application.

Remember that the sheets are easily accessible — the user can always bring them up by making a checkmark on the title line, even if the menu line is turned off. This is particularly useful if you structure your application as a pop-up tool without a menu line, in the mold of a calculator, clock, keyboard, or control panel.

For example, suppose you had a pop-up keyboard tool, and wanted to provide several layouts of keys, and also allow the user to choose either a “realistic” look with raised keycaps on a grey background, or a more compact, lighter “paper-like” look. The place to put the controls would be on an application option sheet, as shown in Figure 67.



**Figure 67: View Sheet for a Keyboard Tool**

## **Issues**

Currently when **Find** and **Spell** are invoked from the **Edit** menu their scope is the entire document. Would it be better if their scope were the current selection?

We're considering adding support for a dynamic **Options** menu that can would contain the names of option sheets for the component that contains the current selection. Is this important?

Currently, if a tool has no menu line (like the keyboard, for example) the only way for the user to access the option sheets is by making a checkmark — there's no visual invitation. We're considering providing a standard button at the right of the title line to bring up the option sheet in this situation. Is this a good idea?

## Chapter 11: Icons

This chapter describes how PenPoint uses application icons, and presents guidelines for using icons, including:

- Guidelines for the graphic design of icons.
- The use of icons to show application state.

## **Application Icons**

Each PenPoint application has an *icon* associated with it. The icon is a small picture that suggests the type of document or the purpose of the tool.

The icon represents the application instance in its closed state. PenPoint displays application icons in several places:

- Icons for full-page documents appear in the notebook Table of Contents.
- Icons for embedded documents or tools are displayed when the document or tool is closed.
- Icons for tools appear in the Tools palette.
- Icons for documents, tools or notebooks appear on the Bookshelf.

The default size for icons throughout PenPoint is 16 X 16 pixels.





Everywhere except the Table of Contents, the user can also select a large size — 32 X 32 pixels — for the icon.

Whether you structure your application as a document, tool, or notebook, you need to provide a bitmap to be used as the application's icon.

The rest of this chapter gives guidelines for the design of icons for both documents and tools. Following these guideline will ensure that your icons fit in with those designed by GO and by other application vendors, thereby helping to bring visual consistency to the multi-vendor PenPoint environment.

## Icons for Documents

Figure 68 shows some typical document icons as they appear in the notebook Table of Contents.

	Notes .....	2
	Memo .....	3
	Drawing .....	4
	Calendar .....	5

**Figure 68: Document Icons in Table of Contents**

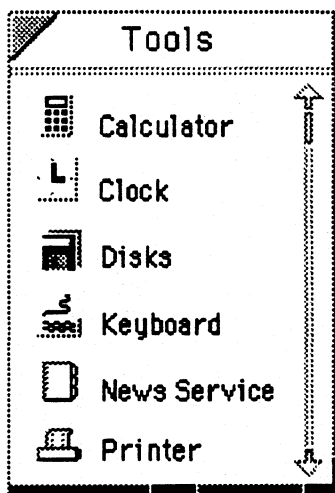
The first icon in the above figure is the default document icon: a blank page. If you don't provide an icon, the default icon will be used.

When possible, show your icon within this standard outline, like the icons for *Memo* and *Drawing*.

The icon for *Calendar* departs from this strict convention, to show a picture that more clearly suggests a multi-page calendar. But it still coherent with the paper metaphor.

## Icons for Tools

Figure 69 shows some typical tool icons as they appear in the Tools Palette.



**Figure 69: Tools Icons in Tool Box**

Note that these icons do not suggest documents. Each suggests a familiar object closely related to the application's function.

The *News Service* application in the above example is structured as a notebook, therefore it has a notebook-like icon.

## Icon Design Guidelines

This section gives guidelines for the graphic design of PenPoint icons.

### Simple Shapes

The most important rule is to keep the picture simple. Don't try to capture too much detail in the 16 X 16 space.

Figure 70 illustrates how simple shapes communicate more directly.



**Figure 70: Keep Icons Simple**

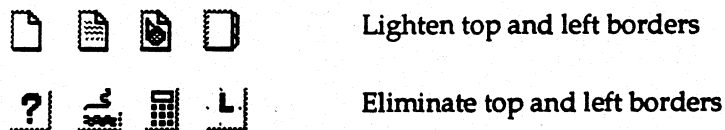
Attempting too much detail makes the icons on the top row busy. Eliminating unnecessary detail, and bringing out a single theme, makes the versions on the bottom row easier to recognize.

### Lightweight Look

The second general guideline is to keep the icons light — primarily white, with black and grey accents.

To help give a lightweight look, and add some visual interest to what are basically small square shapes, PenPoint icons de-emphasize the top and left of the bounding shape.

There are two ways to do this — by using dark grey instead of black for the top and left borders, or by eliminating the top and left borders entirely. Figure 71 shows examples of both techniques.



**Figure 71: Two Ways to Keep Icons Light**

### *Feedback When the Icon is Open*

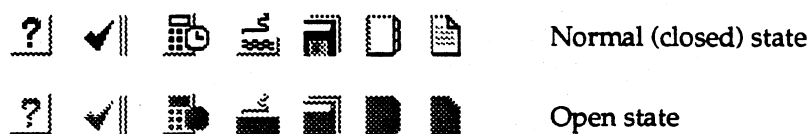
Another consideration in designing the icon is making sure that the user can distinguish between the closed and open states of the icon.

When you create an icon, you define two parts:

- The picture itself, in which each pixel can be either white, light grey, dark grey or black.
- A shape, known as the *mask*, through which the icon will be painted.

When the icon is in its normal (closed) state, each pixel within the mask is painted in the color that was specified when the icon was created. When the icon is in its open state, each pixel within the mask is painted in dark grey. This allows the user to tell at a glance when an icon is open.

Figure 72 shows several icons in both closed and open states.



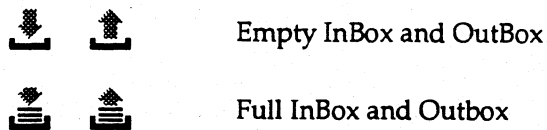
**Figure 72: Open Feedback for Icons**

The above examples show how to design the mask so that the open feedback works well. Notice that the shapes that work best are the ones which are partially open, and partially enclosed, so that the shape of the icon in its normal (closed) state is different enough from the shape in the open state to be clearly distinguished.

## Using Icons to Show Application State

You can give the user useful feedback by changing the icon's picture to reflect the state of your application.

Figure 73 shows how the InBox and OutBox make use of this technique.



**Figure 73: Using Icons to Show State**

The icons in the top row indicate that the the InBox and OutBox are both empty; the icons in the bottom row indicate that InBox and OutBox each contain at least one document.

## Issues



## Chapter 12: Help

This chapter describes the two standard facilities that PenPoint provides for presenting on-line application help:

- *Quick Help* — brief, context-specific help for anything that the user can tap on.
- *The Help Notebook* — for more detailed, procedural help on your application.

## Quick Help.

Tapping the Help icon on the Bookshelf displays the Quick Help Sheet, and puts the input system into *quick help mode*.

As long as the sheet is displayed, the user, tapping anywhere on the screen causes PenPoint to display the help message associated with the object or region under the tap.

The user terminates the mode by dismissing the Quick Help Sheet.

Figure 74 shows the quick help sheet for the column of checkboxes in the Table of Contents that allows the user to turn tabs on and off.

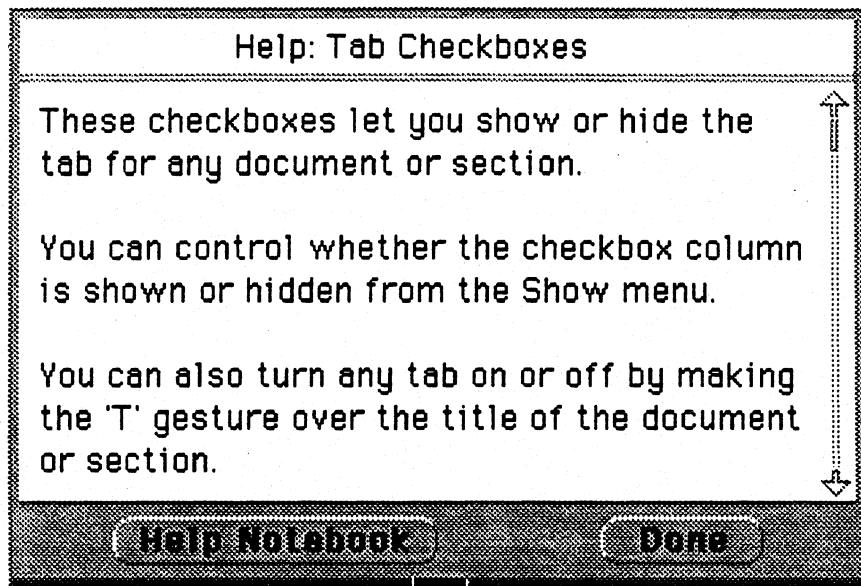


Figure 74: Quick Help Sheet

This example shows quick help for the column of checkboxes in the Table of Contents that allows the user to turn tabs on and off. The text briefly explains the function of the checkboxes, tells how to turn the column off from the Show menu, and mentions the gesture accelerator that forms the other part of the dual command path for this function.

Follow these guidelines when writing quick help messages:

- Keep the message brief. Ideally all the text should fit in the default sheet without requiring scrolling.
- Mention any relevant gesture accelerators.
- Cross-reference to a topic in the help notebook if appropriate.

## Help Notebook

While the Quick Help Sheet is for very brief messages describing objects that the user can see on the screen, the Help Notebook is the place to describe how to use your application.

The user can access the Help Notebook either by tapping a button on the Quick Help Sheet, or by double-tapping the Help icon on the Bookshelf.

The Help Notebook behaves like an ordinary notebook, with the exception that it cannot be resized. This means that you can count on the Help Notebook being a fixed size, and can divide your help text into page-sized chunks.

As part of the installation process, the application's help text is placed in a separate section of the Help Notebook.

## Issues

Is it important to support Hyperlink buttons in the help notebook?

The way Quick Help currently works, it isn't very convenient to browse through menus. The user has to leave the mode each time, then bring up the menu, then re-enter the mode. One way to facilitate browsing is to allow menus to pop up when in Quick Help mode — tapping on the menu label would both give a message about the menu *and* bring up the menu. How important is this?



## Section IV: Processing Pen Input

This section covers issues in processing pen input, including:

- *Processing gestures.* Issues in gesture processing, including how to target gestures, avoiding gesture collisions, complete list of core and non-core gestures, etc.
- *Processing handwriting.* Handwriting processing issues, including the use of translators and constraints.
- *Processing strokes without translating.* Guidelines on using strokes without translating them.
- *User interface for input modes.* Guidelines for allowing the user to control the input mode of your application.



## Chapter 13: Processing Gestures

This chapter covers the use of gestures, including:

- *Gesture targeting*. Guidelines for determining the operand of the operation represented by the gesture.
- *Gesture categories*. Core and non-core gestures.
- *Processing gestures*. How to handle gesture events in an object-oriented environment.

## **Gesture Targeting**

The object that the user intends to operate on is called the *target* of the gesture.

It is not always obvious what the target of a given gesture should be. This section gives guidelines for computing gesture targets.

### *The Target Point*

It is useful to think of each gesture as having a precise *target point* that determines its target. Here are the general rules for determining the target point for a given gesture:

- *Center of Bounding Box.* For some gestures the PenPoint convention is to treat the center of the gesture's bounding box as the target point. These gestures include: cross-out, circle, and all the capital letters.
- *Pen-down.* For a large group of gestures the convention is to treat the point where the pen first touches the screen (the *pen-down point*) as the target point. Such pen-down gestures include: all the taps and flicks, checkmark, pigtail, down-right, up-right, down-left, down-left-flick, down-right-flick, right-up, right-up-flick, and right-down.
- *Visual Focal Point.* When a shape has a strong visual focal point, treat that point as the target point. These gestures include: caret, caret-tap, double-caret, and the arrows.



### *Targetting Unselected Objects*

One of the pen's inherent advantages is that it allows the user to indicate both the operation and the operand in a single, natural gesture.

In order to fully take advantage of this capability, you should allow the user to gesture over objects without first selecting them.

The above rule implies that you need to decide what the most intuitive or useful object is — from the user's perspective — to use as the default target for the operation.

Let's look at a few examples from PenPoint's text component. Some gestures act on characters: tap to select, pigtail to delete. But many editing gestures — X to delete, circle to edit, checkmark to set options, etc. — act on whole words, because the word is the most natural unit of text to edit.

Note that the default target for a given gesture may well differ according to the context. For example, an 'X' deletes a word when made in text, a document or section when made in the notebook Table of Contents, and the entire contents of the field when made in a fill-in field.

### *Targetting the Selection*

In order to allow for the targetting of de-selected objects, gestures should act on the selection *only* if the gesture is made over the selection.

If the selection is very small, it may be difficult for the user to target it accurately. To take an extreme example, a selected lower-case letter 'i' in a 10 point, variable-width font may be only three pixels wide.

To facilitate targetting of such small selections, allow a small "slop zone" around the target point. If this region intersects the selection at all, then take the selection as the target.

In PenPoint, this zone is 3 pixels wide by 5 pixels high. This makes it easy enough, for example, to delete a single character with a pigtail.

### *Targetting Window Objects*

The input system distributes each gesture to the window that the pen first touches when making the gesture.


This presents an implementation problem when the gesture's target point is not the pen-down point, and the object to be targetted is implemented as a separate window.


For example, let's take the common case of making an X to delete the contents of a field in a form. Since the idealized target point of the X is the center of the bounding box, the user will often start the X slightly above the text to be deleted, and end it slightly below. However, if the field is implemented as a separate window, it will never see the gesture, because the input system will distribute it to the window implementing the background of the form.


In such cases, you can obtain good results by implementing a simple algorithm in which the parent window distributes the gesture to the first of its children to be intersected by the gesture.


Note: this "first enter" algorithm is supported by the PenPoint toolkit.


## Core Gestures


- 


*Tap.* This is the most basic gesture. Typically tapping selects an object or pushes a button. Use it for the most basic or important function of an object or region.
- 


*Press-Hold.* (Touch the pen to the screen and pause for a moment.) When done over an object that can be moved by the user, press-hold should always initiate a move of that object. The secondary meaning is, when made not over an object, to sweep out a region for selection. So, for example, when done in text over a selection, press-hold begins a move for the selection; when done in text not over a selection, it begins a wipe-thru selection. Press-hold in the margin of a list should also wipe-thru select, and press-hold on the background of a graphics editor should sweep out a bounding box for selection.
- 


*Tap-Hold.* (Tap the screen once, then touch the pen to the screen again and pause for a moment.) When done over an object that can be copied by the user tap-hold should always begin a copy of that object.
- 


*Flick.* Four directions are distinguished: up, down, left and right. Flicking is used throughout PenPoint to bring more information into view. The user model is that the flick gesture shoves the object in the direction of the flick. Examples include scrolling text, turning notebook pages, exposing overlapping tabs, and cycling through choices in pop-up checklists. For scrolling, flick should shove the object or position under the pen to the opposite boundary of the viewing region — e.g. flicking up in text shoves the line under the pen to the top.
- 


*X.* This is the basic deletion gesture.
- 

*Caret.* Use this for the primary meaning of insert or create in your context. For example, in text the caret brings up a writing pad to insert new text, and in the table of contents the caret brings up a menu allowing you to create a new document or section.
- 

*Circle.* Use for editing the primary attribute of an object. In PenPoint this usually means the objects textual label. The object may be a field in a form, an entry in the table of contents, a title on a document, a tab label, an icon label, a span of text, a spreadsheet cell, etc.
- 

*Checkmark.* Use to display the option sheet for an object. The object may be a document, an accessory, a span of text, an icon, a figure in a graphics editor, a cell in a spreadsheet, etc. This gesture is a workhorse that can almost always carry the burden of changing whatever needs to be changed for an object or region.
- 

*Brackets.* Use to adjust an existing selection in contexts that support the selection of a span of objects (such as text, lists and tables). The left bracket adjusts the starting point of the selection, the right bracket adjusts the endpoint.
- 

*Pigtail.* Use to delete a single character (or the selection) in text.
- 

*Down-Right.* Inserts a space in text. In translated text always inserts a single character, in overwrite boxes the number of spaces depends on the length of the horizontal leg of the L.

## Non-Core Gestures



*Double, Triple and Quadruple Flick.* These gestures — like the single flick — are for “shoving the paper or object to bring more information into view. The effect should correspond to the number of strokes: the more strokes, the farther the shove. PenPoint uses double-flick to scroll to the beginning or end of text, the table of contents, and list boxes, to cycle to the beginning or end of pop-up lists, and to shove overlapping tabs to the top or bottom. Use the triple and quadruple flicks only if you need to provide quick scrolling over a hierarchical object. For example, in a spreadsheet, single-flick should shove the cell under the pen to the view boundary, double flick could scroll to the end of the filled-in cells, and triple-flick could scroll still farther.



*Double, Triple and Quadruple Tap.* In text, these are used to select successively larger units: a word, a sentence or a paragraph. Double-tap is also used to float documents from icons, tabs or Hyperlink buttons. Triple-tap is also used in several places to mean, roughly, “restore the default state.” For example, triple-tapping on a tab makes the tab's label match the title of the document, triple-tapping on the title-line of an option sheet restores the sheet to its default size, and triple-tap on a Hyperlink button links the button to the current selection and updates its label.



*Plus.* Use to mean “toggle selection” in contexts where discrete, discontinuous selections make sense, such as graphics editors or lists. Also used to toggle selection on controls which use tap for invocation, such as icons and Hyperlink buttons. Can also be used in mathematical contexts as an addition sign.



*Square.* Select the area inside the square.



*Circle-Line.* Replace. For text, should bring up an empty editing pad for the object under the gesture.



*Caret-Tap.* Use this for the secondary meaning of insert in your context. In text, for example, caret pops up an insertion pad, and caret-tap creates an embedded insertion pad.



*Circle-Tap.* Create Hyperlink button. This gesture should not be processed by the application.



*Up Arrow and Down Arrow.* Use to make the object larger & smaller, or to zoom up & down.









*Double-Caret.* Create embedded document. This gesture should not be processed by the application. It is intended to be passed through so that it will always pop up the Create Menu from which the user can choose a document to create.



*Check-Tap.* Options for container. This gesture should not be processed by the application. Used primarily to allow the user to display the option sheet for an embedded document whose borders are off.



*Up-Right.* In text, pops up a single-character pad for inserting a single character.

-  *Down-Left*. In text, inserts a paragraph break.
-  *Down-Left-Flick*. In text, inserts a line break.
-  *Down-Right-Flick*. In text, inserts a tab.
-  *Right-Up*. In text, capitalizes the first letter the word, or of each word in the selection.
-  *Right-Up-Flick*. In text, capitalizes the word or selection.
-  *Right-Down*. In text, makes the word or selection lower case.

## Avoiding Gesture Collisions

The term “gesture collision” refers to what happens when two gestures are not distinct enough to be reliably distinguished by the gesture recognition engine.

Gesture collisions are very bad from the user's standpoint, because they cause the system to behave unpredictably. For example, suppose the bracket gesture, meaning extend a text selection, collided with the flick gesture for scrolling. The user would attempt to extend the selection only to find that the display had undergone a radical transformation. Only after the user figured out that the bracket had been mis-recognized as a flick would the situation again make sense.

While collisions can never be completely eliminated, all of the core and non-core PenPoint gestures have been designed, tested and adjusted to minimize collisions.

However, when the 26 capital letters are added to the overall gesture set, many collisions are introduced. This is because the capital letters are recognized by a separate character recognition engine. Character recognition engines are typically tuned for broad coverage of handwriting, and include many prototypes for each letter.

When using capital letter gestures, *it is your responsibility to make sure that the gestures you use don't collide with core or non-core gestures.*

When you find that you want to use a letter accelerator that collides with a core gesture, the core gesture should always take precedence, because it is more widely useful throughout the system.

Let's look at an example. Suppose you want to use the letter C as an accelerator to create a new item in a list. But if the list supports multiple selection, it should support the left and right bracket gestures to adjust spans of selected items. Unfortunately the left bracket collides with the letter C. Adding the C will cause unpredictable behavior, since mis-recognition will occur both ways: C's will be misrecognized as brackets, and brackets as C's.

Because the bracket is a core gesture, you should find another accelerator for the **Create** command, or forgoe the accelerator entirely.

It may be appropriate in some cases to map two gestures onto one operation. Let's look again at the previous example. If the list did *not* support multiple selection, then the brackets would not be needed for the "adjust selection" function. But the potential for mis-recognition would remain high: some percentage of reasonable C's would be mis-recognized as left-brackets. The best solution would be to accept both left-bracket and C as accelerators for **Create**. That way the user would be successful even if he or she made a C that was mis-recognized as a left-bracket.

Note that the set of capital letters is not itself free of collisions — U and V are probably the worst. The same guideline applies: never use such a pair for two different operations. If you do use either of the pair, accept the other letter as well, and map both to the same operation.

A final note of caution. Some collisions are easy to predict from looking at the form of the gesture. But often collisions can not be predicted. The only way to be sure is to test your gesture set on real users.

## **Passing on Unused Gestures**

The general rule for processing gestures is to pass on to your ancestors all gestures that you aren't interested in.

This allows the ancestor to provide standard default behavior when appropriate. For example, caret, caret-tap and double-caret all bring up the **Create** menu by default in class `EmbeddedWin`. The PenPoint text component displays a pop-up writing pad on caret, and an embedded writing pad on caret-tap. By passing double-caret on, it gets the desired default behavior of displaying the **Create** menu.

## **Issues**

Other non-core gestures that GO should provide?

What should be the interpretation of circle and circle-line in non-text contexts?





## Chapter 14: Processing Handwriting

This chapter describes the processing of handwritten characters by the PenPoint handwriting translation system.

Topics covered include:

- *Handwriting translators.* The software objects that perform translation.
- *Constraining translation.* Guidelines for constraining translation for the particular context.
- *Text and numerals.* Handwriting translators can be constrained to recognize text only, numerals only, or both text and numerals.
- *Dictionary.* Optionally, handwriting translators can use the built-in PenPoint dictionary (which the user may have augmented with a personal dictionary) to constrain translation.
- *Templates.* Translators can be further constrained through the use of customized templates.

## **Handwriting Translators**

The translation of handwritten input into ASCII codes is performed by software objects called *handwriting translators*.

Because the handwriting translator and the user interface that mediates between the user and the translator are separate, allowing you to tailor each to suit the needs of your application.

But while these two components are separable, they are also related, and need to be designed together. It is often helpful to the user to have conventions that associate looks with translators. This gives the user a visual cue as to what kind of behavior the field will exhibit.

The standard PenPoint fill-in fields and overwrite fields provide an example of a convention associating a particular kind of translator with a specific visual cue.

When building a translator, the client can choose to 1) provide a grid that the translator will use to segment the input strokes into discrete characters, or 2) tell the translator to use heuristics to perform the segmentation.

The two types of standard PenPoint text fields are designed around this distinction. Overwrite fields present visual segmentation cues to the user, and tell the translator to use that grid. Fill-in fields, on the other hand, present no visual segmentation cues to the user, and tell the translator to determine the segmentation via heuristics.

## Constraining Translation

This section discusses the role of *constraints* in designing handwriting translators.

### *The Problem of Ambiguity*

Even assuming that the translator can correctly segment the input strokes into characters, the problem remains of determining what the user intended in the face of multiple possible valid shapes. Examples of handwritten forms that are indistinguishable in the absence of constraints include the circle (letter 'o' or numeral zero?) and the vertical line (letter 'l' or numeral one?.)

The default behavior of the system is to put like with like. So, for example, a circle preceded by "347" would be translated as a zero, while a circle preceded by "leg" would be translated as the letter 'o'.

### *Improving Translation via Context-Specific Constraints*

The basic technique available to you to deal with this ambiguity is to impose constraints on the interpretation of the input based on its context.

Think of a continuum of input contexts. At one end would be input areas that accept any character that the translation engine is capable of recognizing. An example of such an unconstrained context would be a word processor — there's not much that the designer can say beforehand about what the user is likely to enter.

At the other end of the continuum would be fields that accept only a small, completely specified set of characters. An example of a highly-constrained field would be a social security number, which always takes the form ###-##-###. An even more highly constrained field might only accept a small set of values — for example, a list of states. (In such cases it often makes sense to use an explicit choice list instead of a handwriting translator.)

When building a translator, you can tailor it to the appropriate place on the spectrum. The irreducible design tradeoff is that the more you constrain the context, the more accurate the handwriting recognition will be, and, at the same time, the greater limitation you place on what the user can enter.

The PenPoint translation system permits a great deal of flexibility in constraining translators. The sections that follow describe the different types of constraints and different ways that you can apply those constraints.

### *Constraining to Letters and Numbers*

You can specify that a given translator accept alphabetic letters and symbols, numbers, or both.

### *Constraining Via Dictionary and Templates*

PenPoint allows you to further constrain translators by means of several types of templates:

- *Dictionary.* The dictionary is actually drawn from two sources: the built-in PenPoint dictionary, and the personal dictionary that the user has specified as current in the Installer.
- *Character lists.* A list of ASCII characters.
- *Word lists.* A word list is effectively a small, special-purpose dictionary.
- *Pattern descriptions.* You can describe a pattern by means of a simple language. For example, suppose you define **P** as the set {**A, B, C**}. Then the pattern **###PPP** would accept any digit in the first three positions, and only the letters **A-C** in the second three positions.

Furthermore, you can specify the degree to which each template constrains the translator. There are three levels:

- *Ignore.* The template is ignored during translation.
- *Enable.* The template is used as one of the rules in the process of translation.
- *Veto.* The translator will return *only* words that match the template. If the input does not match the template closely enough, the unrecognized character symbol will be returned for each input character.

### *Guidelines for Using Constraints*

The basic guideline in designing translators is to constrain the input as much as you judge reasonable, given the expected use of the field.

When constraints reflect expected rather than required usage, it is a good idea to give the user some means of turning off the constraints temporarily, to enter characters that don't fit the expected pattern. For example, in a word processor that usually makes use of the dictionary to aid translation, give the user some way to turn off the dictionary when entering a string that he or she knows is not in the dictionary, such as a license plate number.

### **Three Models of Translators and Fields**

As mentioned in a previous section, the translation object is separate from the object used to implement the user interface.

There are three basic architectures to consider when associating translators with fields:

- Each field has its own translator.
- Two or more fields share the same translator.
- One field uses two or more translators, based on a dynamic user setting.

### **Issues**



## Chapter 15: Processing Strokes Without Translating

This chapter describes how untranslated strokes can be used.

Topics covered include:

- *Markup Layers*. Allowing the user to markup a document.
- *Ink as a Data Type*. The use of untranslated scribbles as a data type.
- *Signature pads*. Standard component.
- *Deferred translation in forms*.
- *Deferred translation in notetaking applications*.

Note: This chapter to come.

**Issues**





## Chapter 16: Presenting Input Modes

This chapter describes how to allow the user to switch between input modes.

Topics covered include:

- Types of input modes
- Mode control via pop-up list at right of menu line
- Mode control via palette at right of menu line
- Mode control via palette line
- Mode control via palette in scroll margin

## **Translation Modes**

Drawing and painting applications in traditional graphical user interfaces typically use different input modes for selecting, erasing, and various flavors of drawing and painting.

In addition to these familiar modes, you may want to organize your application around different *translation modes*.

The most common translation modes are:

- *Gestures*. This is an edit mode, in which the user can edit or scroll via gestures in the work area.
- *Sketch*. In this mode the user's strokes are recorded without translation, as free-form sketches.
- *Shapes*. In this mode common shapes are recognized: circles, squares and ellipses.
- *Text*. In this mode strokes are processed by the handwriting translation system.

This list is not exhaustive. For example, you might have modes that recognize specialized shapes, such as musical notes in a composing program.

Also, note that translation modes are not necessarily mutually exclusive.

### **Always Show the Current Mode**

If you do provide different input modes, it is essential to provide visual feedback so the user can tell at a glance what the current mode is.

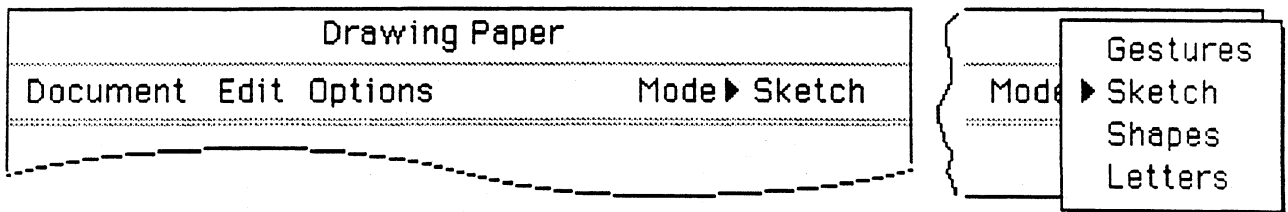
The remainder of this chapter describes different ways to present mode controls.

### Mode Control on Menu Line

Given the importance the mode control, it is a good idea to place the mode control where it is always visible to the user. One obvious place for it is on the menu line.

#### *Pop-up List at Right of Menu Line*

If the input modes are exclusive, you can meet the requirement to show the current mode by using a pop-up list, as shown in Figure 75.

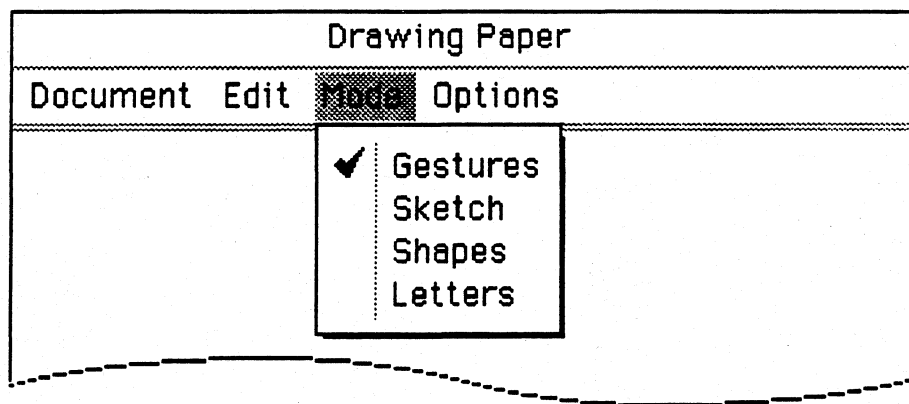


**Figure 75: Pop-up List for Exclusive Modes**

Put the list at the right of the menu line, to separate it visually from the other menus.

Using a pop-up list has the advantage that it allows for the compact presentation of many of choices.

Putting the mode control in a standard menu, as shown in Figure 76, is not recommended.

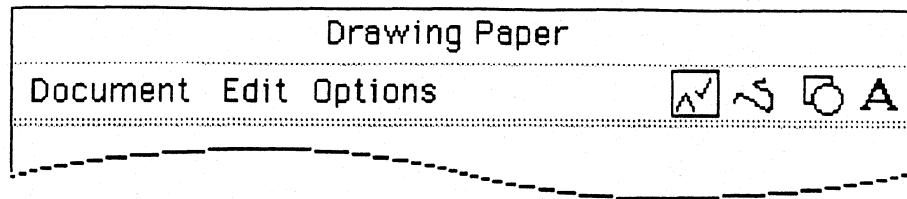


**Figure 76: Menu with Checklist for Modes (Not Recommended)**

This usage should be avoided, because it forces the user to display the menu to discover what the current mode is.

*Palette at Right of Menu Line*

You can also use a palette for the mode control, as shown in Figure 77 below.

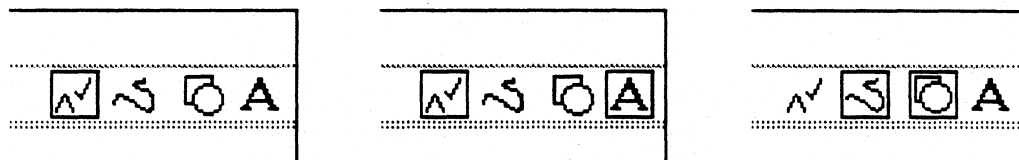


**Figure 77: Palette for Exclusive Modes**

When the number of choices is small, the palette is just as compact as the pop-up list, and is easier to use because the user can switch to any mode with a single tap.

If the modes are additive rather than mutually exclusive, it is essential to show them all at once, so the user can see which are on at any given moment.

The palette is the right control for this situation, as shown in Figure 78.



**Figure 78: Palette for Non-Exclusive Modes**

### Mode Control in Palette Line

You can also put the palette in a separate palette line. Do this if you have more modes than will fit at the right of the menu line

In Figure 79 below the mode control has been combined with other controls on a palette line.

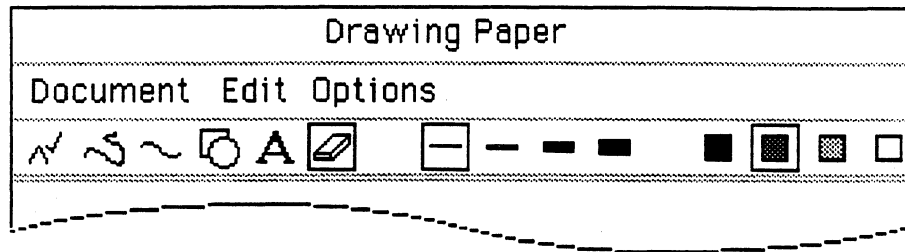


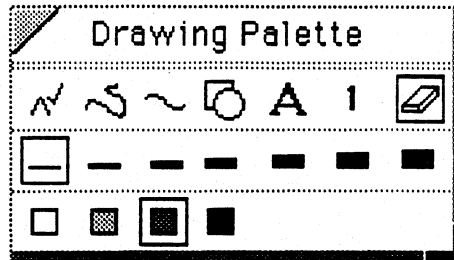
Figure 79: Palette Line

The line in the above example contains three groups of controls: input modes at the left, line thicknesses in the center, and fill colors on the right.

If you use a palette line, make sure to add a control to the standard Access Sheet to allow the user to toggle the palette line on and off.

### **Mode Control on Pop-up Sheet**

In addition, it is often appropriate to put the mode control on a pop-up palette or option sheet. Figure 80 shows a pop-up palette for a drawing application.



**Figure 80: Mode Control on Pop-up Palette**

In the above example, the first line of the palette controls the input mode, the second line controls the line thickness, and the third line controls the the fill color.

### **Issues**

## Chapter 17: Handling Keyboard Input

This chapter describes the use of the keyboard.

Note: This chapter to come.





# Section V: Manipulating Application Objects

This section describes selecting and editing of objects within applications.

Topics covered include:

- *Selection.* User interface conventions for allowing the user to manage the current selection.
- *Move/Copy.* Drag and Drop model.
- *Handwriting Processing.* Standard PenPoint editing pads and text gestures.
- *Text Editing.* Standard PenPoint editing pads and text gestures.



## Chapter 18: Selection

This chapter gives guidelines for managing the system-wide *current selection*.

Topics covered include:

- *User Model*. Definition of selection from the user's viewpoint.
- *Selection Feedback*. Standard selection feedback for different types of objects.
- *Selecting and Deselecting*. Selecting a single object. Adjusting an existing selection. Selecting groups of objects. Selection of discontinuous multiple objects. Auto-selection of gesture targets.
- *Text Selection*. Pending-delete selection. Selection behavior when the primary input device is the pen vs. the keyboard.
- *Option Sheet Selection*. Handling option sheets that themselves contain objects that take the selection.

## User Model

Graphical user interfaces in the Xerox Star tradition generally present the user with what has been called the "noun-verb" model for invoking operations. In this model the user first identifies the object of interest (the noun) and then indicates which operation to apply to that object (the verb).

The act of identifying the operand is called *selecting an object*, or *making a selection*, and the object, once selected, is called the *current selection*.

## Selection Feedback

The current selection must be distinguished visually from the other objects on the screen so that the user can identify it at a glance.

While it is important for selection feedback to be as consistent as possible, no one convention suffices for all types of objects. Figure 81 shows selection feedback for four different object types.

Four score and **seven** years ago.

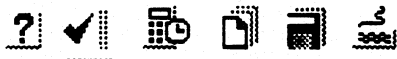
*Text.* Selected span is rendered on grey background.

A Clockwork Orange

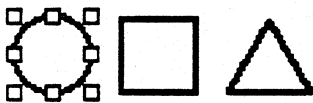
**Planet of the Apes**

China Blue

*Lists.* Selected item rendered on grey background. The background usually extends the full width of the list.



*Icons.* Selected icon is underlined.



*Resizable figures.* Selected figure is indicated by the presence of resize handles at the corners and at the midpoint of the edges of the figure's bounding rectangle.

**Figure 81: Selection Feedback for Different Object Types**

One rule to follow is to avoid using inversion as selection feedback, because gestures that the user makes will not be visible over the black mass of the selection.

## Selecting and Deselecting

This section describes the PenPoint conventions for selecting and deselecting objects.

The importance of selection is reflected in the fact that all mouse-based interfaces use the main mouse button to select objects. Those interfaces that came out of the Xerox tradition devote two mouse buttons to selecting — one to making new selections and the other to adjusting existing selections.

One of the strengths of the pen is that it permits the user to easily make gestures other than tapping. PenPoint takes advantage of this strength by providing a variety of complementary ways to make and adjust selections. Selection-related gestures include tap, double-tap, plus, press-hold-drag, and left and right brackets.

### *Selecting a Single Object*

The basic gesture for selecting a single object is tap. Tapping in text should select a character, tapping a figure in a drawing should select the figure, tapping an item in a list should select the item.

Some types of objects function like controls, in that they use the tap gesture to invoke their main operation. So, for example, tapping on an icon opens the icon, and tapping on a tab or a hyperlink button turns to the associated document.

In these “control-like” cases — where selection is not as important as the object's primary function — you should interpret plus as the selection gesture.

## *Manipulating Application Objects*

### *Deselecting an Existing Selection*

From the user's perspective, the current selection is only useful with reference to an operation that the user intends to make in the very near future. At other times, it may appear as an artifact that is irrelevant, or even distracting.

Therefore it is important that the user be able to easily deselect any object. You should always allow the user to deselect in two ways:

- *Tapping on the object.* Tapping (or making the plus gesture) on an existing selection should deselect it.
- *Tapping on the background.* Tapping anywhere on the background of the region should also deselect any existing selection. Examples of include the background of a drawing region, or the dotted lines connecting document names with their page numbers in the notebook Table of Contents.

### *Extending a Selection by Dragging*

In many contexts the user can select multiple objects — a span of text, a block of items in a list, a block of cells in a spreadsheet, several figures in a drawing, etc.

If your application supports multiple selection, you should allow the user to select adjacent objects via the press-hold gesture. The user touches the screen, waits for the press-hold timeout to elapse, and then drags out the region to select with the pen.

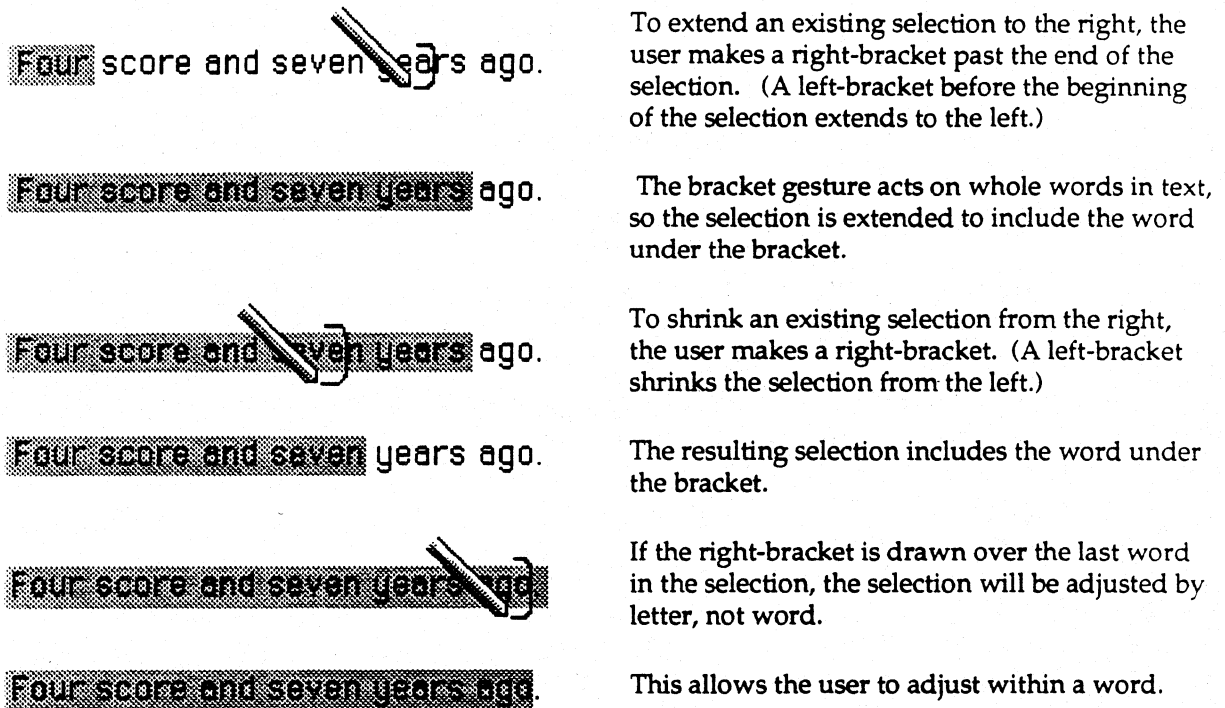
Use one of these two conventions for providing feedback during the drag:

- *Wipe-thru*. If your application uses a light grey background to indicate selection (e.g. text and lists) jump the grey (from letter to letter, or item to item) as the user drags. This is often called *making a wipe-thru selection*.
- *Bounding Box*. If your application uses some form of outline or underline to indicate selection (e.g. graphic figures) use a bounding box to provide feedback as the user drags.

*Extending a Selection With Brackets*

The left and right bracket gestures should adjust an existing selection in any formatted context such as text, lists, or tables.

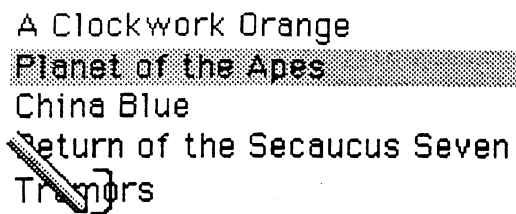
Figure 82 shows how the brackets work in text.



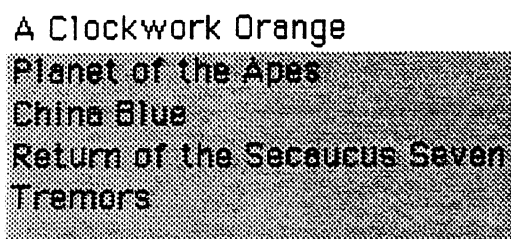
**Figure 82: Adjusting a Text Selection with Brackets**



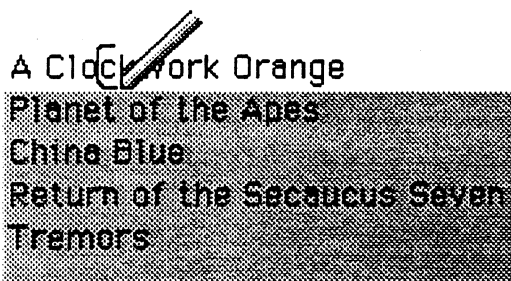
Figure 83 shows how the brackets work in a list.



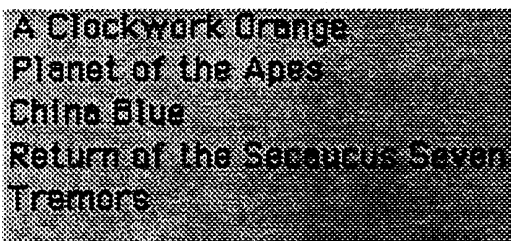
To extend an existing selection to include items following the selection, the user makes a right-bracket.



The selection extends to include the item under the bracket.



To extend an existing selection to include items preceding the selection, the user makes a left bracket.



The selection extends to include the item under the bracket.

**Figure 83: Adjusting a List Selection with Brackets**

## *Manipulating Application Objects*

### *Discontiguous Multiple Selections*

The previous section described how to allow the user to select multiple objects that are adjacent.

It is often helpful to the user to allow the selection of multiple objects that are not adjacent. This is referred to as *discontiguous selection*. Discontiguous selection is commonly supported for lists and applications dealing with graphic objects.

If you do support discontiguous selection, use the plus gesture as a toggle: a plus over an unselected object should select it, a plus over a selected object should deselect it.

### *Auto-selection of Gesture Targets*

Because the pen allows the user to indicate the operand as well as the operation in one gesture, the PenPoint user need always select the object to be operated on.

For example, the user can draw an 'X' to delete an object, a circle to edit a label, a checkmark to bring up an object's option sheet, all without first selecting the object.

Often such a gesture results in a popup sheet, pad or note — an edit pad for a document in the Table of Contents, an option sheet for an icon, a proof menu for a word, a confirmation note for deleting a document, etc. In all such cases you should programmatically select the object targeted by the gesture before displaying the pop-up, in order to provide a visual relationship between the pop-up and its target.

## Text Selection Issues

This section discusses a couple of text-related issues.

### *Pending Delete*

Text selection follows the “pending-delete” de-facto standard, in which text entered via the keyboard replaces any existing text selection.

### *Primary Input Preference*

Text should observe **Primary Input** preference, and follow the convention used in PenPoint's text component. If the user has set the preference to **Pen**, then tapping selects a single character.

If the preference is set to **Keyboard**, then tapping sets an I-beam indicating the insertion point for keyboard input. This mode allows for compatibility with traditional, mouse- and keyboard-based graphical user interfaces.

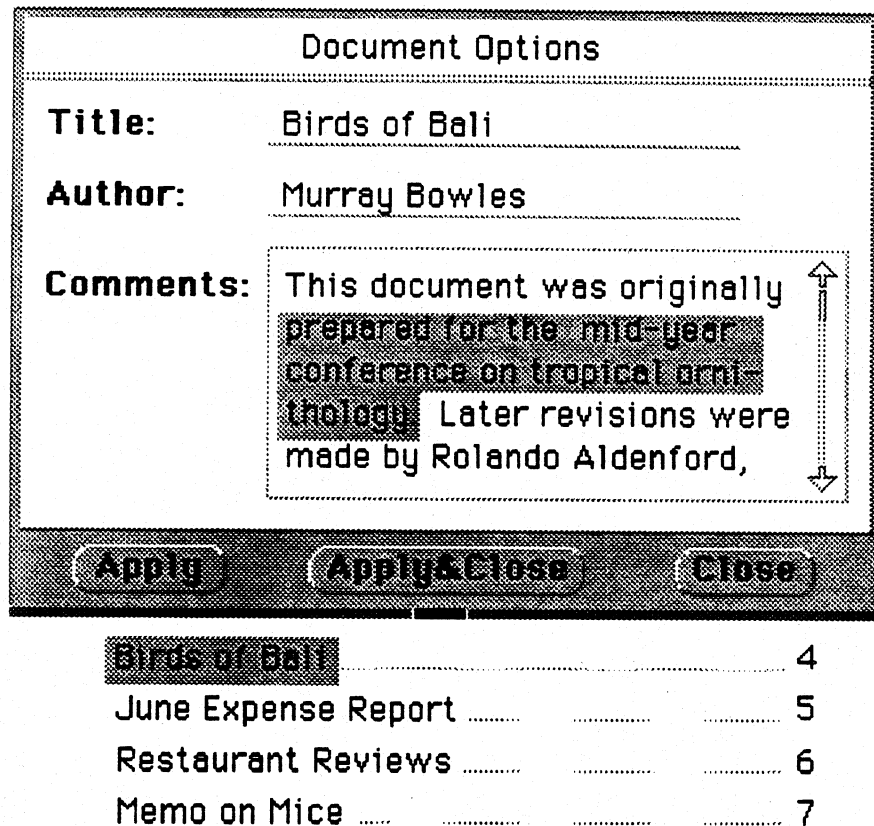
## Selection in Option Sheets

Because PenPoint option sheets are modeless, the user can change the selection while the option sheet is displayed. And option sheets can contain components that themselves accept the selection — such as fields or a text boxes.

This presents a problem in a single-selection model: if the user makes a selection within the option sheet, the object associated with the option sheet will be deselected. When that happens the Apply button on the option sheet will become inactive.

In order to solve this problem PenPoint supports the concept of a *preserved selection*. You need to specify that the existing selection remain selected when the user makes a selection within an option sheet. That will allow the option sheet's command buttons to remain active.

Figure 84 shows illustrates this dual selection.



**Figure 84: Option Sheet Selection**

The figure shows a document option sheet. The document whose options are being shown has remained selected, even though the user has selected some text in the **Comments** box.

## Chapter 19: Move/Copy

This chapter gives guidelines for implementing the Move and Copy operations.

Topics covered include:

- User Model.
- Detailed description of both the move and the copy process.
- Inter-application data transfer issues.
- Step-by-step examples.

## **User Model: Drag and Drop**

This section describes the PenPoint user model for moving and copying objects, and contrasts it with the model presented by traditional, mouse-based interfaces.

### *Move and Copy in Mouse-based Interfaces*

Mouse-based graphical user interfaces present the user with two very different models for moving and copying objects, the direct manipulation *drag and drop* model and the clipboard-based *cut/copy/paste* model.

The drag and drop model is simple and direct: the user drags the object to the desired location with the mouse. It works well for objects such as icons on the work surface or figures in a graphics editor.

But, as traditionally used, the drag and drop method has severe limitations:

- The application designer must map it to either move or copy, since there's no way for the user to distinguish which operation is desired.
- It doesn't work for all types of objects — text, for example.
- It doesn't work when the source and destination are not in view at the same time.
- It doesn't work when the source and destination are in separate applications.
- It doesn't work for deferred or repeated operations.

Because of these limitations, traditional graphical user interfaces supplement the drag and drop method with the clipboard-based method that supports the cut, copy and paste operations.

### *Move and Copy in PenPoint*

PenPoint presents a single user model — drag and drop — for moving and copying objects throughout the environment.

The PenPoint drag and drop method addresses the limitations described in the previous section:

- It provides a way for the user to distinguish move from copy.
- It works for all types of objects — cons on the Bookshelf, entries in the Table of Contents, text in a word processor, figures in a graphics editor, cells in a spreadsheet, etc.
- It works when the source and destination are not in view at the same time.
- It works for transferring data across document boundaries, and between different types of documents.
- It supports deferred operations. The cut and paste model allows the user to transfer data to a special clipboard, and copy it from the clipboard as needed.

In order to fully deliver the benefits of this unified model to the user, it is very important that all applications follow the same conventions for implementing move and copy.

## **Support for Implementing Move and Copy**

This section breaks the move and copy operations down into the steps of invoking the operation, dragging the object and targetting the destination.

It describes both the simple case in which the source and destination are both in the same document and both visible on the screen, and variations (source is offscreen, destination offscreen, deferred transfer, etc.)

This standard user interface is supported by a protocol in class embedded win.

### **Initiating the Move or Copy Operation**

Before dragging the object, the user must signal to the system whether the operation is a move or a copy. Following the dual command path principle, PenPoint provides two ways to initiate the operation:

- *Menu.* The user first selects the object, then chooses Move or Copy from the Edit menu.
- *Gesture.* The gesture for Move is *press-hold*: the user touches the object with the pen and pauses for a moment. The Copy gesture is *tap-press-hold*: the user taps the object with the pen, then touches again and pauses.

### *Move and Copy Marquee*

After the user initiates the operation, the system signals that the object is ready to be dragged by surrounding it with an animated dashed outline, called a *marquee*.

The *move marquee* is a single outline; the *copy marquee* is double. This makes it clear to the user throughout the process whether the drag will result in a move or a copy.



### Drag Icons

In implementation terms, what actually happens when the move or copy marquee begins is that a marqueeed *drag icon* is displayed over the selected object.

This section describes how to display the drag icons in various situations.

Ideally, the drag icon should contain an exact rendering of the selected object, so that it appears to the user as if the marquee simply begins around the selected object. Figure 85 gives several examples.

Four score  and years ago.

A Clockwork Orange

 Planet of the Apes

China Blue

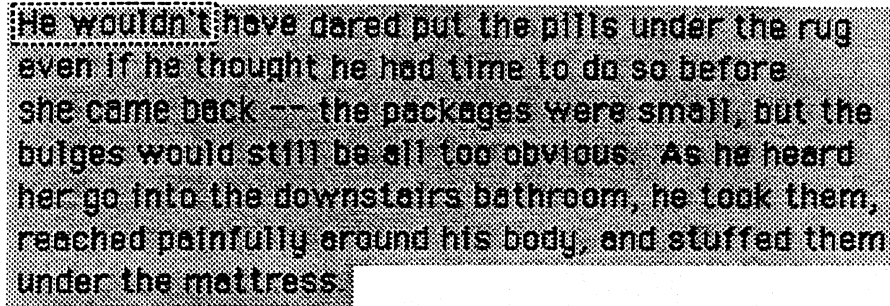


**Figure 85: Drag Icons that Mimic the Selected Object**

PenPoint supports this by automatically copying the portion of the screen in the selected region to the icon.

It is not always possible to copy the bits from the screen into the icon.

One problem is that the selected object may be too large to drag — a selected text span, for example, might well be larger than the screen itself. If the objects in your application can be so large that they are impractical to drag, you should limit the size of the drag icon, as shown in Figure 86.



**Figure 86: Move Marquee for Large Text Selection**

In the above example, the user has selected an entire paragraph, and then chosen Move from the Edit menu. The application has determined that the selection spans more than one line, and so has limited the move marquee to a rectangle at the beginning of the paragraph.

The user can also invoke move or copy from the menu when the selection is scrolled entirely offscreen.

To support this scenario, PenPoint provides standard drag icons for both move and copy, as shown in Figure 87.



**Figure 87: Standard Drag Icons for Move and Copy**

When the selection is entirely offscreen, display the appropriate drag icon in the center of the application's client area.

Another situation in which the standard drag icons are handy is for dragging figures in a drawing program across document boundaries. (See Example #6 later in this chapter.)

## Completing the Operation

After initiating the move or copy, the user completes the operation by dragging the drag icon to the destination with the pen. There are several variations on the process:

- If the user begins with a gesture, and the destination is already visible on the screen, the user simply drags as soon as the marquee begins.
- If the user begins with a gesture, and the destination is *not* already visible on the screen, the user lifts the pen after the marquee begins, which leaves the drag icon “floating” on the screen. At this point the user brings the destination into view by scrolling or turning to another document, then drags the icon to the destination.
- If the user begins from the menu, then the drag icon appears over the selected object, and the user then drags it to the destination.

### *Drag Rectangle*

As the user drags, a dashed outline the size of the object — called the *drag rectangle* — follows the pen.

The drag rectangle, like the marquee surrounding the object, is single or double to indicate move or copy.

### *Targetting the Destination*

When the destination allows free-form positioning of objects — as is usually the case in drawing or painting programs — the drag rectangle allows the user to position the object at the desired destination. When the user lifts the pen, the object should appear precisely where the outline was.

In such situations, the drag outline should remain in the same position relative to the pen as when the pen touched the object.

Often the destination application imposes constraints on the positioning of objects — as, for example, in text, lists and tables.

The user model for targetting the destination in such formatted contexts is to point the pen at the white space between two adjacent objects. In text, this means pointing between two words or two paragraphs. In a list, it means pointing between two items in the list. In a table, it means pointing at the border dividing two rows or columns.

A usability problem may arise here if the user is unclear whether to target by reference to the drag rectangle or the pen tip. For example, in text, should the user point the pen or the left edge of the drag rectangle at the destination?

To avoid this problem, when the user begins dragging in text, snap the drag outline so that the left edge is under the pen tip. That way the user will succeed using either the pen tip or the drag rectangle as the reference point for targetting.

Note that because the snapping of the drag rectangle must be done when the user begins dragging away from the move or copy icon, the decision as to whether or not to snap the rectangle must be based on whether or not the *source* application is formatted.

### *Canceling the Drag*

The user can cancel the move or copy operation at any point short of completion.

When the drag icon is floating (and the pen is not touching the screen) the user can cancel the operation by tapping the drag icon.

Changing the selection by selecting another object anywhere in the system will also cancel the operation and dismiss the drag icon.

The user can cancel even after beginning to drag, by dragging back over the drag icon and lifting the pen. Or, the user can drag and drop on the title-line or menu line. The icon will jump to the new position and remain floating, and the user can then tap it to cancel.

## **Moving and Copying Between Applications**

This section discusses implementation issues related to inter-application data transfer.

### *Standard Data Types*

To facilitate the transfer of data between applications, there is a set of standard data formats that all PenPoint applications should read and write:

- ASCII
- Microsoft RTF (Rich Text Format)
- TIFF
- Object

**Note:** This list of data formats is preliminary, and will be defined further before the release of PenPoint 1.0.

### *Object Data Transfer*

Ideally, user should be able to move or copy information anywhere in the PenPoint environment, without regard to the type of the source and destination or the type of data being transferred.

To facilitate this, there's a special data transfer type called *object*.

Any region that supports embedding within it will always — even if it doesn't take the specific data type — accept an object that holds the data. Therefore you should support the rendering of your data as an object.

This effectively gives the whole system the functionality of a visual clipboard that accepts any number of objects. For example, the user can select a paragraph and drag it down to the document's cork margin, where it will remain as an icon (see example #5 in the next section.) Then the user can either move it or copy it at any time, as desired.

### *Regions that Don't Accept Embeddees*

When the user drops a drag icon onto a destination that can't accept the type of data represented by the icon, and won't accept embedded objects of any type, the destination should refuse to accept the data. The drag icon will jump to the location of the drop and remain floating.

This happens automatically, so most applications will not have to worry about it. The title and menu lines are examples of areas that will not accept any type of data, so users can always "park" a drag icon over them.

### **Examples**

The next several pages give step-by-step examples illustrating moving and copying in several common scenarios.

*Manipulating Application Objects*

*Example 1: Moving a Word in Text*

Four score **seven** and years ago.

The user first selects the word to be moved.

Four score **seven** and years ago.

Then the user initiates the move with the press-hold gesture.

Four score **seven** and years ago.

When the user drags the pen, the marquee outline jumps so that the pen tip is centered on the left edge of the outline.

Four score **seven** and **years** ago.

The user points the pen tip directly at the desired location — in the gap between two words.

Four score and **seven** years ago.

When the user lifts the pen, the move is completed and the text reformatted. The newly moved word remains selected.

**Figure 88: Moving a Word in Text**



Example 2: Moving an Item in a List

A Clockwork Orange  
Planet of the Apes  
China Blue  
Return of the Secaucus Seven  
Tremors

The user first selects the item to be moved.

A Clockwork Orange  
Planet of the Apes  
China Blue  
Return of the Secaucus Seven  
Tremors

Then the user initiates the move with the press-hold gesture.

A Clockwork Orange  
Planet of the Apes  
China Blue  
Return of the Secaucus Seven  
Tremors

The user drags out the dashed outline, until the pen-tip points between the rows where the item is to be moved.

A Clockwork Orange  
China Blue  
Return of the Secaucus Seven  
Planet of the Apes  
Tremors

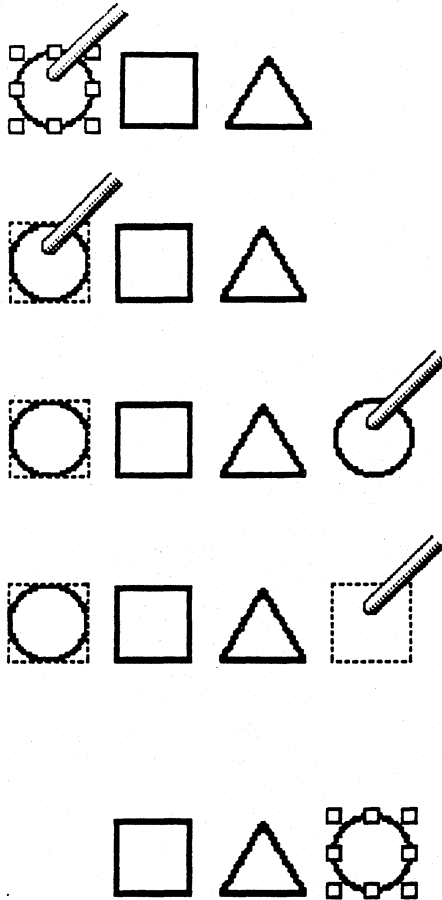
When the user lifts the pen, the move is completed and the list redisplayed.

The newly-moved item remains selected.

Figure 89. Moving an Item in a List

## Manipulating Application Objects

### Example 3: Moving a Figure



The user first selects the figure to be moved.

The eight resize handles indicate that the figure is selected.

Then the user initiates the move with the press-hold gesture.

When the user drags the pen, the application can either:

- 1) drag a rendition of the object itself, or
- 2) drag a dashed outline of the selection.

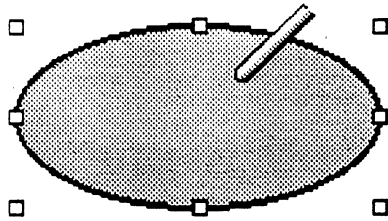
If an outline is dragged, it should *not* jump relative to the pen. It should also be the exact size of the selection. This allows the user to use the outline as a guide for positioning the object precisely.

When the user lifts the pen, the move is completed.

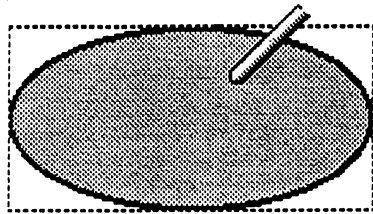
The newly moved object remains selected.

**Figure 90: Moving a Figure**

Example 4: Moving a Figure to a Far Destination

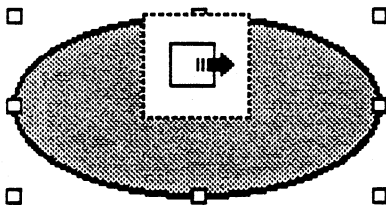


The user first selects the figure (or figures) to be moved.



Then the user touches the selected figure with the pen and pauses until the move marquee appears.

(At this point, the user can drag to a local destination without lifting the pen, as described in the previous figure.)

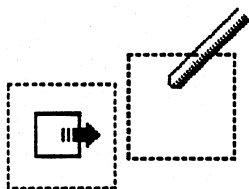


The user lifts the pen at this point, causing the *move icon* to appear.

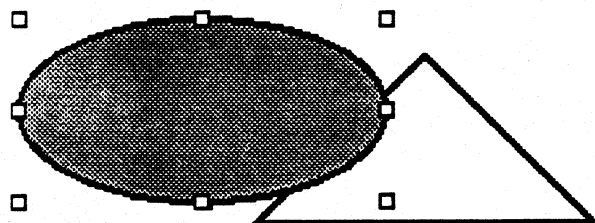
The move icon has a glyph symbolizing the move operation, and is surrounded by the move marquee. It is always the same size, regardless of the size of the selected object.



While the move icon is displayed, the user brings the destination into view by turning to another document (or scrolling within the same document.)



When the destination is in view, the user continues the drag by touching the move icon and dragging out a dashed outline.

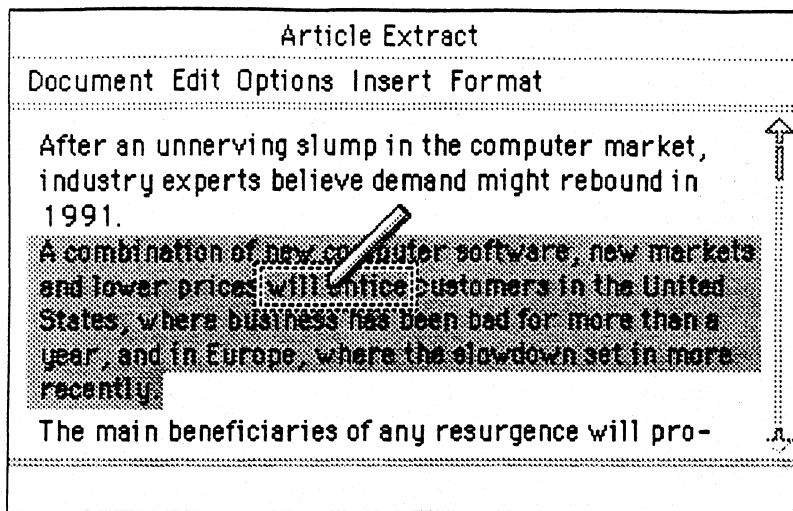


When the user lifts the pen, the data transfer takes place, and the object appears at the destination.

The object is placed in the same location relative to the pen as when the move was initiated.

Figure 91: Moving a Figure to a Far Destination

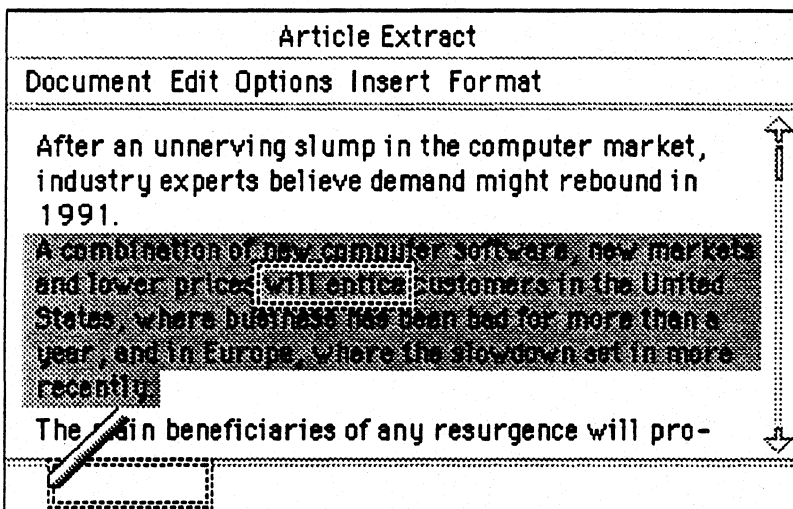
Example 5: Object Data Type: Copying a Paragraph to the Bookshelf



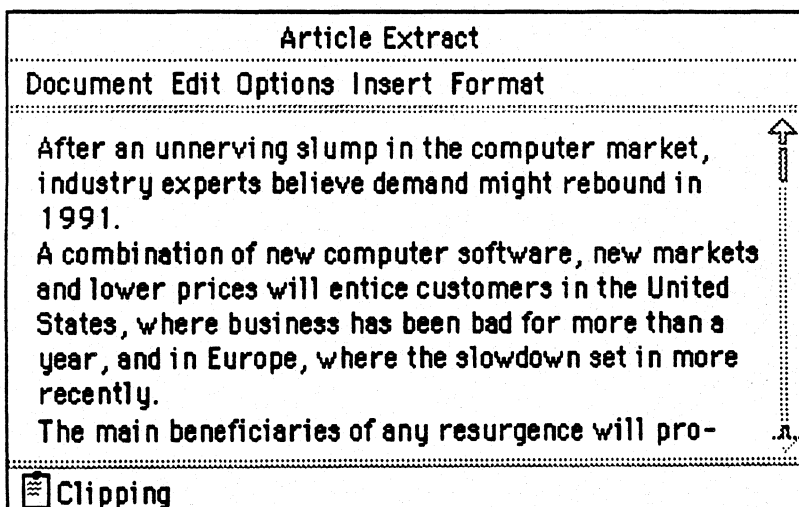
The user first selects the paragraph then begins the copy with the tap-p gesture.

The copy marquee appears around

Note: The drag icon is limited to a 1 around the pen, because the selectic than one line.



The user drags to the Cork Margin.



The user lifts the pen, dropping the onto the Cork Margin.

Because the Cork Margin doesn't ac directly, the data is transferred in th format. It appears as an icon in the

Figure 92: Copying a Paragraph to the Bookshelf

**Issues**



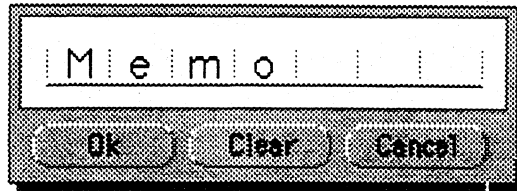
## Chapter 20: Editing Text

This chapter gives guidelines for editing text in several contexts: overwrite fields, fill-in fields, and full text views.

## Pop-up Edit Pads

PenPoint provides a standard type of pop-up pad specialized for editing fields and labels.

Figure 93 shows an edit pad.



**Figure 93: Pop-up Edit Pad**

### *Usage by PenPoint*

These edit pads are used extensively in PenPoint to provide a uniform way for the user to edit fields and labels of all sorts, including:

- document titles on the document
- document and section names in the Table of Contents
- file names in the Disk Viewer
- icon labels
- tab labels
- words and phrases in the text component

### *Usage by Applications*

Unless the context of your application requires a specialized type of input pad, you should use the standard edit pads for editable fields and labels.

In order to make editing in your application as convenient and efficient as possible, you may want to design special editing pads that are further specialized to the types of objects in your application.

In designing these pads, remember the principle of the dual command path, and allow the user to enter information either through the handwriting engine (gestures or characters) or by tapping (buttons, menus and scrolling lists).



### *Invoking Edit Pads*

The circle is the core PenPoint gesture for the basic **Edit** operation. Drawing a circle should always bring up the editing pad for the object under the circle.

Tapping should also display the edit pad, unless there is a more important operation to map to tap. For example, tapping on a fill-in field in a dialog or option sheet displays the edit pad for the field, since editing is the primary operation on the field. But tapping on a tab or Hyperlink button turns to the associated document, and tapping on a title in the Table of Contents selects the document. In these cases editing is not the primary operation.

The object being edited should take the selection, so that the user can see at a glance what the target of the pad is.

The text being edited should appear in the pad's overwrite boxes, with a couple of blank boxes at the end to allow the user to add one or two characters without resizing the pad.

### *Input Behavior*

The standard edit pads are modal — while they are displayed input to other parts of the screen is blocked, and they automatically take the focus of keyboard input.

If you build a specialized edit pad, with more functionality, it may make sense for the pad to be modeless, like the PenPoint option and dialog sheets, so that the user can leave it up and use it to edit several objects.

### *Size Preference*

Pads should observe preferences — if you create one and re-use it, you should check preference each time the pad is displayed.

## *Manipulating Application Objects*

### *Using Edit Pads*

Edit pads have two different modes of behavior with respect to translation:

- *Input mode:* Whenever there are no boxes with translated characters, the pad is considered to be in input mode. The user writes into the boxes, and no translation occurs until the user taps **OK**.
- *Edit mode:* When there is at least one box containing a translated character, then the user's input is translated automatically when the user lifts the pen out of proximity to the screen for more than a moment.

This behavior is designed to support the most common usage scenario in which the user first enters the word or phrase into a blank pad, then taps **OK** to translate, then overwrites the characters as needed to correct any translation errors.

The pads also accept three editing gestures, described under *Gestures in Character Boxes*, below.

The edit pad buttons work as follows:

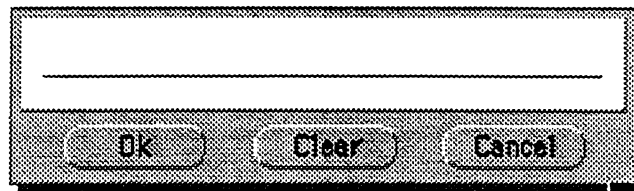
- **OK:** If there are no boxes with translated characters, tapping **OK** translates the contents of all the boxes. If one or more boxes contain translated characters, then tapping **OK** returns the contents of the pad to the caller and dismisses the pad.
- **Clear:** Clears the pad.
- **Cancel:** Dismisses the pad without making any change to the object being edited.

## Pop-up Writing Pads

The caret gesture should pop up a writing pad to allow the user to insert new text.

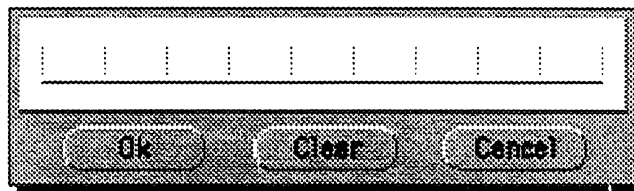
Before displaying a pop-up writing pad, you should check the state of the **Pad Style** option on the Handwriting Preference Sheet, and display the appropriate form of pad.

Figure 94 shows a ruled pad.



**Figure 94 Pop-up Writing Pad (Ruled)**

Figure 95 shows a boxed pad.



**Figure 95: Pop-up Writing Pad (Boxed)**

## *Manipulating Application Objects*

### **Gestures in Character Boxes**

Character boxes accept only three editing gestures, as described below.



*Tap-Hold.* Makes the box the focus of keyboard input.




*Pigtail.* Deletes the character in the box.





*Down-Right.* Inserts one or more spaces, depending on the length of the horizontal line.


## Gestures in Fill-in Fields


For the Developer's Release of PenPoint, the standard text fields accept the following gestures:


- 


*Tap.* Observes the **Preferred Input** preference. In **Pen** mode (the default), pops up an edit pad containing the field's contents. In **Keyboard** mode, sets the I-beam.
- 


*Double-tap.* Toggles selection of entire contents of field.
- 


*Press-Hold.* If field contents are not selected sets the I-beam. If field contents are selected, initiates move of field contents.
- 


*Tap-Hold.* If field contents are selected, initiates copy of field contents.
- 

*X.* Deletes the contents of the field.
- 

*Scratchout.* Deletes any word touched by the gesture.
- 

*Circle.* Pops up an edit pad containing the field's contents.
- 
















*Caret.* Pops up an empty writing pad targeted at the tip of the caret. The pad is either ruled or boxed, depending on the setting of the **Pad Style** preference.
- 

*Pigtail.* Deletes the character under the pen-down point.
- 

*Down-Right.* Inserts a single space at the pen-down point.

## Gestures in Text Views

In addition to the core gestures, the text component accepts the following gestures:

-  *Double Tap.* Select word.
-  *Triple Tap,* Select sentence.
-  *Double Flick (4 directions).* Scroll to the beginning, end, left edge or right edge.
-  *Circle-Line.* Replace. Bring up an empty editing pad for the word or selection.
-  *Caret-Tap.* Create an embedded insertion pad.
-  *Circle-Tap.* Create Hyperlink button.
-  *Up Arrow and Down Arrow.* Increase/decrease point size for the word or selection by the increments in the character option sheet.
-  *Up-Right.* Insert a single character.
-  *Down-Left.* Insert a paragraph break.
-  *Down-Left-Flick.* Insert a line break.
-  *Down-Right-Flick.* Insert a tab.
-  *Right-Up.* Capitalize the first letter the word (or of each word in the selection.)
-  *Right-Up-Flick.* Capitalize the word or selection.
-  *Right-Down.* Make the word or selection lower case.
-  *Double-Caret.* Create embedded document. Pops up the Create Menu.

- ✓. *Check-Tap*. Brings up option sheet for the document.
- B** *B*. Make the word or selection bold.
- F** *F*. Bring up find sheet, set to start the search from the point of the gesture.
- I** *I*. Italicize the word or selection.
- P** *P*. Proof word.
- N** *N*. Make the word or selection “normal” — i.e. turn off bold, italic, and underlined attributes.
- S** *S*. Bring up spell sheet, beginning checking from the point of the gesture.
- U** *U*. Underline the word or selection.





## Chapter 21: Forms

Forms are one of the most important classes of applications for PenPoint. This chapter gives guidelines for designing and supporting forms.

Topics covered include:

- *Emulating paper forms.* Allowing the user to markup a document.
- *Designing forms from scratch.* The use of untranslated scribbles as a data type. Shrink to fit area.
- *Validation in forms.* Allowing the user to markup a document.

Note: This chapter to come.



## Section VI: Application Architecture

This section describes the several basic ways a PenPoint application can be structured:

- *Document*: A page in the user notebook.
- *Tool*: An accessory that pops up or takes over the entire screen.
- *Section*: A section in a notebook.
- *Notebook*: A separate notebook.
- *Service*: Inbox services, Outbox services, and background servers.
- *Components*: Inbox services, Outbox services, and background servers.

Note: This entire section is rough.



## Chapter 22: Documents

This chapter describes the several basic ways a PenPoint application can be structured:

- When to structure your application as a document.
- User model: document on a single notebook page.
- Basic document components and layout.
- Local page controls.
- Handling embedding within documents.

## **When to Structure Your Application as a Document**

If your application makes sense as something that could be found in a notebook, present it as a document.

This is intended to be a broad category, and include not only such obvious documents as a word-processor, drawing program, or form, but also applications that would not ordinarily be thought of as documents in the traditional sense, such as database front ends, games, etc.

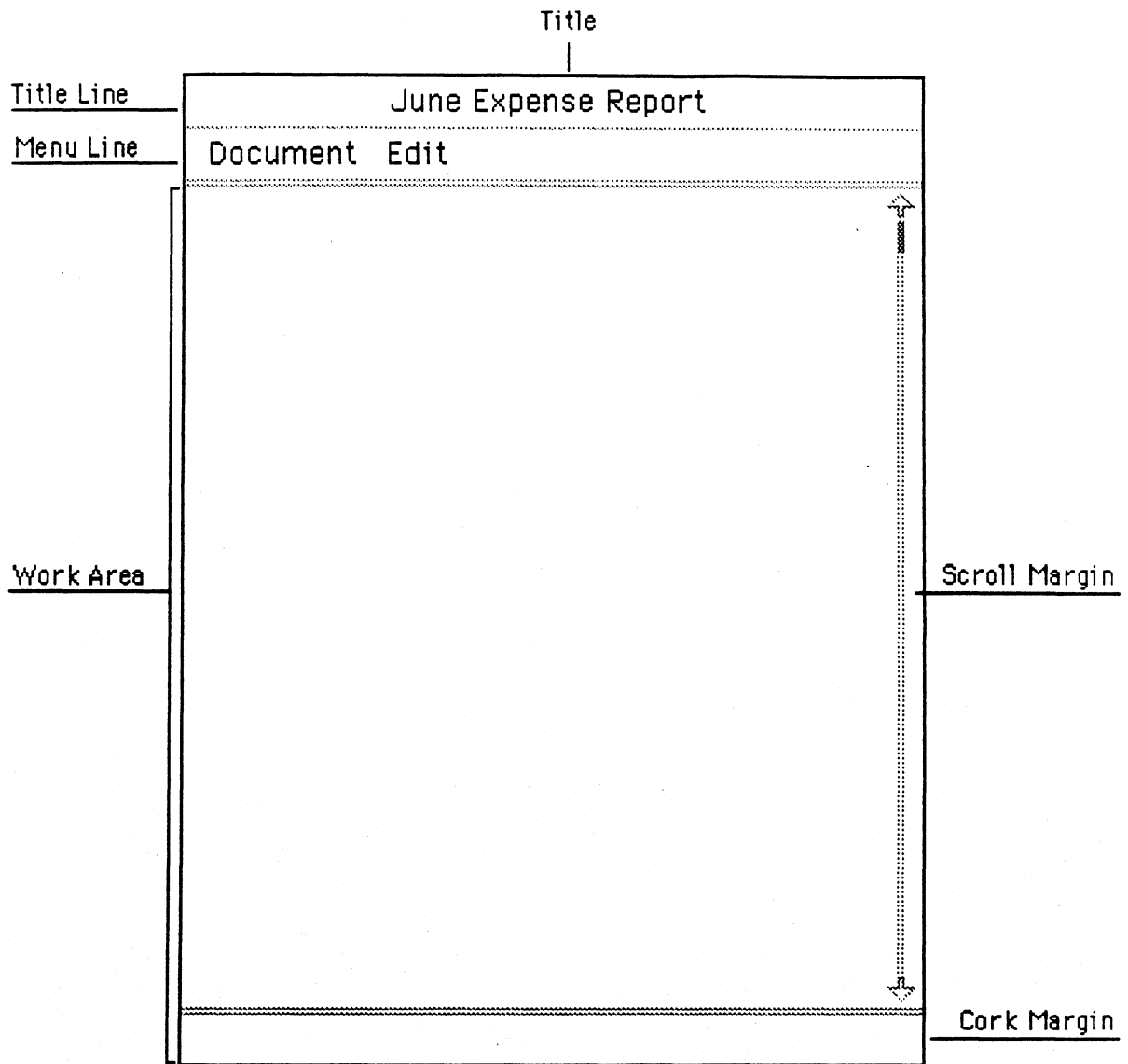
### **User Model: Document on Single Notebook Page**

The basic user model for the Notebook User Interface is that each page of the notebook contains a single document.

The document itself may extend across many printed pages. The user can scroll the offscreen portions of the document into view by using the scroll margins or the flick gestures.

### Basic Document Layout

Figure 96 shows the basic components and layout for documents.



**Figure 96: Basic Document Layout**

These components are described briefly on the following page.

The standard document components are:

- *Title Line.* Contains the title of the document.
- *Menu Line.* Each label in the menu line represents a category of commands. The user can show or hide the menu line at any time from the document option sheet.
- *Scroll Margin.* Each document takes up one page of the notebook, but the document itself may be of any length. The scroll margin allows the user to bring offscreen portions of the document into view. The user can show or hide the scroll margin at any time from the document option sheet.
- *Cork Margin.* An margin below the body of the document into which the user can place icons (useful for pop-up notes or annotations) and link buttons (useful for scrolling directly to specific places in the document or for linking to other documents). The cork margin is not shown by default; the user can turn it on at any time from the document option sheet.
- *Work Area.* The bulk of the page is reserved for the information in the document itself.

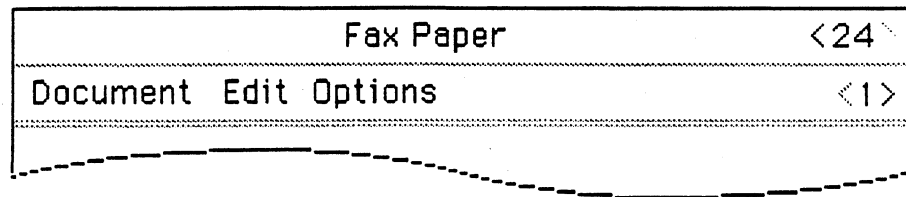


## Local Page Controls

If your application is naturally structured as a series of pages, it may be appropriate to provide a visible control for paging within the document. The next two sections describe the two standard ways to do this.

### *Local Page Numbers*

If your document has numbered pages, put the page control at the right of the menu line, directly below the notebook page number, as shown in Figure 97.

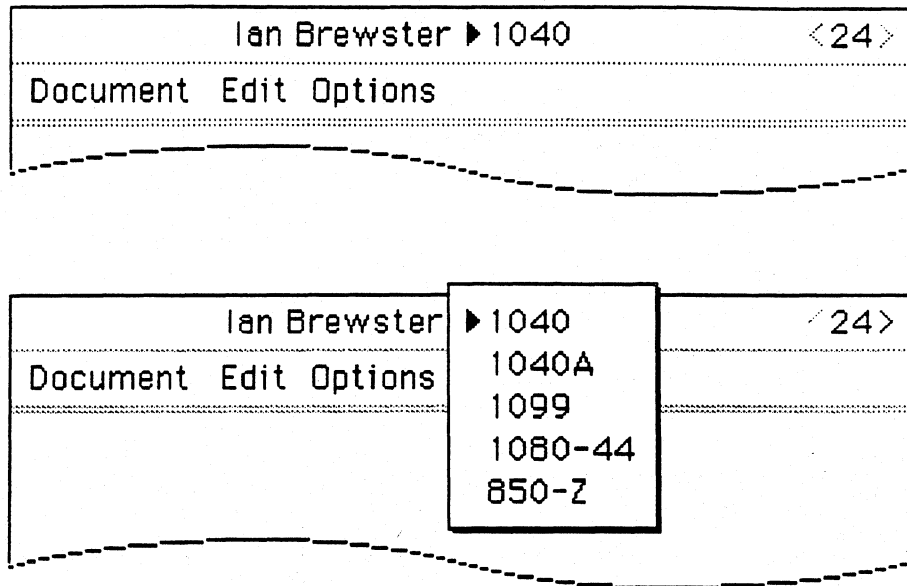


**Figure 97: Local Page Number on Menu Line**

The page control is a standard toolkit control; tapping on the left bracket turns to the previous page, tapping on the right bracket turns to the next page. Pressing either of the brackets flips through pages.

Pop-up List on Title Line

As an alternative to the local page number control, you can use a pop-up list on the title line, as shown in Figure 98.



**Figure 98: Pop-up List on Title Line**

The pop-up list is appropriate when there are a limited number of named pages.

## Managing Embedded Objects

One of the important innovative features of PenPoint is its support for live application embedding. This section describes the different ways your application can make use of this feature.

Applications can be divided into two categories with respect to how they handle embedded objects: *transparent embeddors* which rely entirely on the Application Framework to handle embedded objects, and *child-aware embeddors*, which take embedded objects into account when laying out their data.

### *Transparent Embeddors*

Transparent embeddors don't open up holes for their embeddees.

An example would be a simple drawing application which did not contain any commands to arrange the objects within it, or snap them to a grid.

The embeddees are clipped to the parent, and travel with the parent as it is moved or copied. The embeddees don't count in the notebook page numbering sequence, and don't show up in the notebook Table of Contents.

### *Child-aware Embeddors*

Child-aware embeddors pick up the embedding protocol and modify the presentation of their contents based on their children. In short, they make room for their children.

Most applications will fall into this category. Examples include:

- A word processor that makes room for embeddees, either by treating the embeddee as one huge character, or in some other way, such as flowing text around it.
- A drawing program that allows the user to snap objects — including embeddees — to a grid.
- A corkboard-like container that has commands to arrange objects within it according to various layouts.
- An application such as a spreadsheet, that presents its data in tabular format. Since opening a space for the embeddee to be opened in-line doesn't make sense in the context of a table, the embeddor can restrict its embeddees to open in pop-up format, overlapping the work area.

Each of these behaviors requires that the embeddor do some work. For example, the word processor needs to re-layout its contents when the embeddee is opened, closed or resized.

You need to think about this at the beginning of the application design process, so your data structures support embedded objects. For example, a spreadsheet would need to be able to store an object as well as a formula in a given cell.

### **Issues**

## Chapter 23: Tools

This chapter describes structuring applications as tools. Topics include:

- When to structure your application as a tool.
- Pop-up and fullscreen tools.

## *Application Architecture*

If your application doesn't fit well into the form of the notebook, present it as a tool that the user finds as an icon in the Tools Palette. When the user opens the icon, the tool either pops up over the notebook, or takes over the entire screen.

Examples of pop-up tools include clocks, calculators, a snapshot program, etc.

If you need to control the layout of the tool exactly, you can specify that it always be full-screen, and not resizable. An example of a full-screen tool is the PenPoint Handwriting Customization application.

It is often straightforward to decide whether an application should be a document or an accessory. In some cases, however, it may be appropriate for an application to take both forms.

Take, for example, the PenPoint Disk Manager. Its primary usage is as a pop-up tool that the user opens from the Bookshelf, uses, and closes. But the user can also copy an instance of the Disk Manager into a notebook. This supports the usage scenario of having multiple disk manager instances on multiple notebook pages, each of which stay open to a particular location on a particular disk.

## **Issues**

Should tools save their state when they are opened/closed/opened?

## **Chapter 24: Component/Application Model**

This chapter describes structuring applications components and wrappers.

## *Application Architecture*

One approach you can take in PenPoint is to structure your application in two parts: a visual component that does most of the work of presenting information to the user, and an application "wrapper" that provides some other functionality such as a menu line.

The component can then be used in several applications.

The Minitext application bundled with PenPoint is structured on the component/application model.

The component needs to be self contained. None of the basic behavior should be implemented by the wrapper.

### **Issues**

We don't have a well-known place for component resources -- so, in practice for now, the application needs to know about the component.



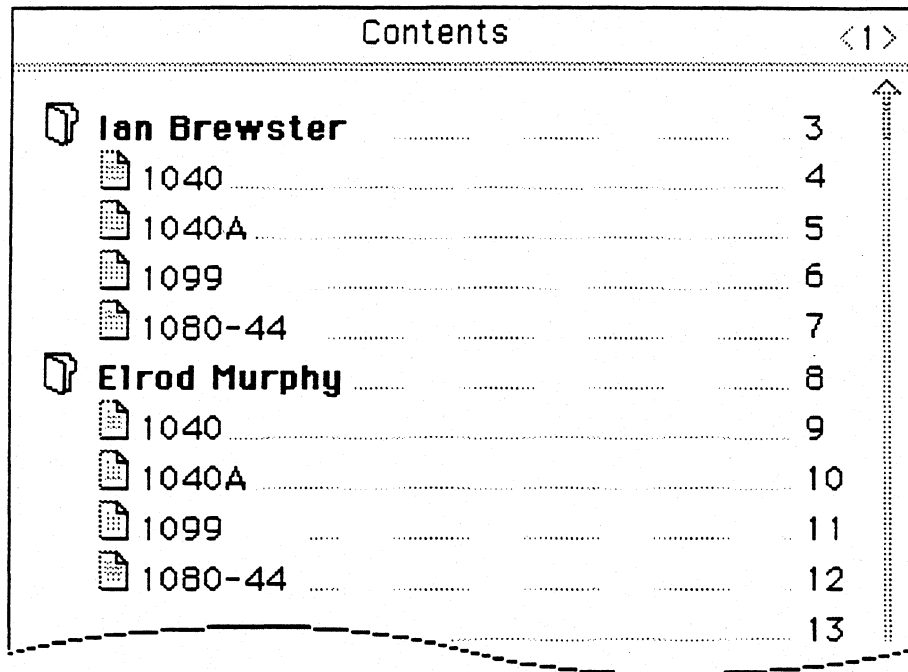
## **Chapter 25: Notebooks and Notebook Sections**











This chapter describes structuring applications as sections in the user notebook, and as complete notebooks.

## Structuring Your Application as a Section

You can structure your application as a group of separate documents, organized into a notebook section that manages them. In this architecture each notebook section embodies an instance of the application.

Figure 99 below shows an example.



Contents		<1>
 <b>Ian Brewster</b>	.....	3
 1040	.....	4
 1040A	.....	5
 1099	.....	6
 1080-44	.....	7
 <b>Elrod Murphy</b>	.....	8
 1040	.....	9
 1040A	.....	10
 1099	.....	11
 1080-44	.....	12
	.....	13

**Figure 99: Application as Section**

In the example above, each section contains the tax forms for a different client.

This application architecture has advantages and disadvantages. It fits in well with the notebook metaphor, but it is harder to program and coordinate across the separate processes on each notebook page.

## **Subclassing the Table of Contents**

While you can use the normal Table of Contents and treat the section as nothing more than a group of normal documents that can travel together, you can also subclass the Table of Contents to tailor its functionality or user interface.

### *Subclassing to Provide Inter-document Functionality*

You may want to provide control and coordination functions across the documents on the different pages — e.g. summing totals of forms, cross-form fill-in or validation.

### *Subclassing to Provide a Customized User Interface*

For example, suppose an insurance application had a section with separate documents for different sections of the car that has been damaged. From the Table of Contents, the section would look normal. But the section itself would show a picture of a car, with hotspots on the car for turning to each page in the section.

## **Structuring Your Application as a Notebook**

You can also structure your application as a separate notebook.

This is appropriate if your application is a vehicle for delivering information that makes sense in a notebook format. The PenPoint Help Notebook is an example of an application that is structured as a notebook.

You can either specialize the notebook by subclassing or you can just use the ordinary notebook as a delivery vehicle.

## **Issues**

## Chapter 26: Services

This chapter describes conventions for services that interact with the InBox and OutBox.

Topics include:

- *Transfer services (Inbox & Outbox).*
- *Subclassing Inbox and Outbox sections.*
- *Non-transfer services.*
- *Tracking the user data font preference.*

## *Application Architecture*

There are two types of services:

- *Transfer services:* Transfer services manage either the sending of information through the Outbox, or the receiving of information through the Inbox, or both.
- *Non-transfer services:* Non-transfer services include such background services as database servers.

## **Issues**

## Chapter 27: Installation of Applications and Resources

This chapter covers issues related to the installation process, and also gives guidelines for installable resources such as fonts.

This section describes the building blocks PenPoint provides to build your user interface.

Topics include:

- *Installation process.* The installer. Configuring the application.
- *Stationery notebook.* For documents. Pre-loaded stationery.
- *Tools Palette.* For tools.

When the user installs your application, the PenPoint Installer places an instance of the application in either the Stationery Notebook, or the Tools Palette, depending on whether you have specified that the application is a document or a tool.

If you specify that your application is a document, the installation process results in the creation of a section in the Stationery Notebook named for the application. The section contains at least one document — also named for the application — which is known as the application's *default stationery*. You can specify that other stationery be automatically created for the user as part of the installation process.

If you specify that your application is a tool, the installation process results in an icon of the application being placed in the Tools Palette.

You can also specify multiple tools for a given application.

## Issues