1994

Designing a reusable class cluster: a hypertext cluster

Anoop Kumar Trivedi
University of Wollongong

UNIVERSITY OF WOLLONGONG

COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

This work is copyright. Apart from any use permitted under the Copyright Act 1968, no part of this work may be reproduced by any process, nor may any other exclusive right be exercised, without the permission of the author.

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.

Recommended Citation

Designing a Reusable Class Cluster: A Hypertext Cluster

A Thesis submitted in partial fulfilment of
the requirements for
the award of Master of Science (Hons.)

from
The University of Wollongong

by
Anoop Kumar Trivedi, B.Tech.
Department Of Computer Science

March 1994
I hereby declare that I am the sole author of this thesis. I also declare that the material presented within is my own work, except where duly acknowledged, and that I am not aware of any similar work either prior to this thesis or currently being pursued.

Anoop K. Trivedi
March 1994
Abstract

This thesis is a contribution to the study of object-oriented software engineering, focusing upon the reuse-approach to build reliable and extendable software. A hypertext system has been developed to study the reuse.

The ET++ and MacApp framework class libraries provide standard components that can be re-used to develop application-specific programs. A plug-compatible design is presented in this work; this design approach facilitates building applications that work with frameworks based on similar principles running on different platforms.

Hypertext is chosen as an example domain for designing class clusters, while ET++ and MacApp are selected as frameworks. A survey of reuse and hypertext is done at various levels. The hypertext specific classes are designed, keeping the commonalities in mind to provide adequate generalization. These classes are implemented on the ET++ and MacApp frameworks. There are variations in the two implementations, but overall design of the classes remains the same. The design is validated by testing it against the changing requirements of the system and adding new functionality to the system. The effectiveness of the plug-compatible approach is explored.
Preface

I would like to express my sincere thanks to my supervisor Assoc. Professor Neil. A. B. Gray for his help, advice, and encouragement throughout the course of this research. Every discussion with him left me full of new ideas to explore.

Acknowledgments are also due to Gordon Cheng for his guidance during this project and for helping me to grip the ET++ and MacApp environments.

I would like to thank David I. Raymond and Graham Meintjes for their assistance in the preparation of this document.

This work would not have been possible without the use of the ET++ framework which proves free software can be reliable.

No amount of thanks can be enough to express my warmest appreciation to my brother for his love and support throughout my study as a Master’s student.
# Contents

**Preface**

1 **Introduction**

1.1 Motivation ................................................................. 7
1.2 Objectives ................................................................. 8
1.3 Outline ........................................................................... 9

2 **Software Reusability and Hypertext - A Literature Survey** 11

2.1 Overview ................................................................. 11
2.2 Reusability - Background ............................................. 11
   2.2.1 Birth of Software Reusability ...................................... 12
   2.2.2 Reuse Programs .................................................... 13
   2.2.3 Changing Definition of Reuse ...................................... 14
2.3 Object-Oriented Paradigm ............................................. 15
   2.3.1 The Principles of Object Oriented Technology .............. 16
2.4 Object-Oriented Reuse ............................................... 19
   2.4.1 Mechanisms of Reuse ........................................ 19
   2.4.2 Examples of Reuse ............................................... 20
2.5 Class Libraries ........................................................ 21
2.6 The Problem Domain ................................................ 24
   2.6.1 Problems with Code Reuse ...................................... 25
   2.6.2 Reusable Clusters ............................................... 26
   2.6.3 Design to Plug Approach ...................................... 26
2.7 The Application Domain ............................................. 28
   2.7.1 Why Hypertext as a Cluster? ................................. 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.8 Hypertext - A Survey</td>
<td>29</td>
</tr>
<tr>
<td>2.9 The Milestones</td>
<td>31</td>
</tr>
<tr>
<td>2.10 Components of Hypertext Systems</td>
<td>34</td>
</tr>
<tr>
<td>2.11 Summary</td>
<td>38</td>
</tr>
<tr>
<td>3 Hypertext System: Analysis and Design</td>
<td>39</td>
</tr>
<tr>
<td>3.1 Hypertext Requirements</td>
<td>39</td>
</tr>
<tr>
<td>3.2 ET++ and MacApp: Application Frameworks</td>
<td>43</td>
</tr>
<tr>
<td>3.2.1 The Command-handler classes</td>
<td>46</td>
</tr>
<tr>
<td>3.2.2 Building a program on framework library</td>
<td>47</td>
</tr>
<tr>
<td>3.2.3 MacApp</td>
<td>53</td>
</tr>
<tr>
<td>3.2.4 ET++</td>
<td>55</td>
</tr>
<tr>
<td>3.3 Hypertext: Analysis and Design</td>
<td>57</td>
</tr>
<tr>
<td>3.3.1 A Data Model for Hypertext</td>
<td>57</td>
</tr>
<tr>
<td>3.3.2 Views for Hypertext</td>
<td>62</td>
</tr>
<tr>
<td>3.3.3 Application and Document classes</td>
<td>67</td>
</tr>
<tr>
<td>3.4 Interaction among hypertext objects</td>
<td>70</td>
</tr>
<tr>
<td>3.4.1 Interaction among application and document objects</td>
<td>70</td>
</tr>
<tr>
<td>3.4.2 Interaction among document, view and data model classes</td>
<td>71</td>
</tr>
<tr>
<td>3.4.3 Handling menu and mouse in hypertext system</td>
<td>71</td>
</tr>
<tr>
<td>3.5 Other useful classes</td>
<td>76</td>
</tr>
<tr>
<td>3.5.1 Palette class cluster</td>
<td>76</td>
</tr>
<tr>
<td>3.5.2 Dialog classes</td>
<td>82</td>
</tr>
<tr>
<td>3.5.3 Container and I/O classes</td>
<td>83</td>
</tr>
<tr>
<td>3.6 Summary</td>
<td>86</td>
</tr>
<tr>
<td>4 Implementation and Evaluation</td>
<td>87</td>
</tr>
<tr>
<td>4.1 Overview</td>
<td>87</td>
</tr>
<tr>
<td>4.2 ET++ based Hypertext implementation</td>
<td>87</td>
</tr>
<tr>
<td>4.2.1 Command handler classes implementation</td>
<td>88</td>
</tr>
<tr>
<td>4.2.2 Hypertext data-model classes implementation</td>
<td>99</td>
</tr>
<tr>
<td>4.2.3 I/O, Container, and Dialog classes</td>
<td>103</td>
</tr>
</tbody>
</table>
## List of Figures

2.1 NIHCL "Collection" classes - A part NIHCL class library. ................................ 22  
2.2 A cluster of classes providing some service to the clients. ............................. 27  
2.3 The Adaptor Approach. .................................................................................. 27  
2.4 A Hypertext Node linked with another node with help of marks. ................. 35  

3.1 Nodes, links and structure of a file. ............................................................... 40  
3.2 The marking mechanism. ................................................................................ 41  
3.3 Text and picture node directories. ................................................................. 42  
3.4 MacApp Class Library - Some of the popular classes. ................................. 44  
3.5 ET++ Class Library - Some of the popular classes ........................................ 45  
3.6 An executing program and some of the "command-handler" objects that will be present. ................................................................. 48  
3.7 Object diagram showing document saving and loading mechanisms. .......... 49  
3.8 Object diagram showing window view mechanism. ...................................... 50  
3.9 Object diagram showing window mouse mechanism. ................................... 51  
3.10 Object diagram showing menu command chain mechanism. ....................... 53  
3.11 The "Command Handler" classes from MacApp library. .............................. 54  
3.12 The "Container" classes from ET++ library. ................................................. 55  
3.13 The "Command Handler" classes from ET++ library. ................................ 56  
3.14 Initial design for class HTNode. ................................................................. 58  
3.15 The tree structure of node classes. ............................................................... 59  
3.16 The HTextNode and HTPicNode classes of HTNode cluster. ...................... 59  
3.17 A cluster of link classes - HTLink and HTLinkManager classes. ............... 61  
3.18 Interaction between node and link objects. .................................................. 61  
3.19 A cluster of mark classes - HTMark, HTextMark and HTPicMark classes. 63
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.20</td>
<td>HTTextView class.</td>
<td>63</td>
</tr>
<tr>
<td>3.21</td>
<td>The view components of a picture based editor.</td>
<td>64</td>
</tr>
<tr>
<td>3.22</td>
<td>A cluster of picture view classes - HTPicView and HTPaletteView classes.</td>
<td>65</td>
</tr>
<tr>
<td>3.23</td>
<td>The Browser interface windows.</td>
<td>66</td>
</tr>
<tr>
<td>3.24</td>
<td>Class diagram for HTBrowserView.</td>
<td>66</td>
</tr>
<tr>
<td>3.25</td>
<td>Inheritance diagram of class HTApplication.</td>
<td>68</td>
</tr>
<tr>
<td>3.26</td>
<td>Class HTDocument.</td>
<td>69</td>
</tr>
<tr>
<td>3.27</td>
<td>Object diagram showing interaction between HTDocument and HTApplication classes.</td>
<td>71</td>
</tr>
<tr>
<td>3.28</td>
<td>Object diagram showing interaction between document, view and data classes.</td>
<td>72</td>
</tr>
<tr>
<td>3.29</td>
<td>Hypertext text editor and its menu-bar.</td>
<td>73</td>
</tr>
<tr>
<td>3.30</td>
<td>Object diagram showing various actions that can be taken a user using a menu-bar.</td>
<td>74</td>
</tr>
<tr>
<td>3.31</td>
<td>Hypertext picture editor.</td>
<td>75</td>
</tr>
<tr>
<td>3.32</td>
<td>Class HTPaletteView.</td>
<td>76</td>
</tr>
<tr>
<td>3.33</td>
<td>A cluster of HTPaletteItem classes.</td>
<td>77</td>
</tr>
<tr>
<td>3.34</td>
<td>Class HTPalManager.</td>
<td>78</td>
</tr>
<tr>
<td>3.35</td>
<td>Handling mouse mechanism in picture editor.</td>
<td>79</td>
</tr>
<tr>
<td>3.36</td>
<td>Marking and Dragging mechanism in a picture editor.</td>
<td>79</td>
</tr>
<tr>
<td>3.37</td>
<td>Hypertext specific command classes.</td>
<td>80</td>
</tr>
<tr>
<td>3.38</td>
<td>Interaction mechanism among palette view and command objects.</td>
<td>81</td>
</tr>
<tr>
<td>3.39</td>
<td>Dialog view classes and dialog boxes.</td>
<td>82</td>
</tr>
<tr>
<td>3.40</td>
<td>A Cluster of I/O classes.</td>
<td>84</td>
</tr>
<tr>
<td>3.41</td>
<td>HTFileBuffer class and object diagram.</td>
<td>86</td>
</tr>
<tr>
<td>4.1</td>
<td>The main command handlers for hypertext program.</td>
<td>88</td>
</tr>
<tr>
<td>4.2</td>
<td>The hypertext editors and browser.</td>
<td>89</td>
</tr>
<tr>
<td>4.3</td>
<td>Command object classes in hypertext program.</td>
<td>98</td>
</tr>
<tr>
<td>4.4</td>
<td>Implementation of hypertext text editor and browser.</td>
<td>107</td>
</tr>
<tr>
<td>4.5</td>
<td>Implementation of hypertext picture editor.</td>
<td>108</td>
</tr>
<tr>
<td>4.6</td>
<td>Design diagram for classes THTApplication and THTDocument.</td>
<td>111</td>
</tr>
</tbody>
</table>
4.7 Design diagram for classes THTPaletteView and THTPictureView. . . . 114
4.8 Class diagram for MacApp command classes cluster. ................. 117
4.9 Class diagram for MacApp dialog classes. .............................. 119
4.10 Screen dump from MacApp example of hypertext based text editor. . 121
4.11 Screen dump from MacApp example of hypertext based picture editor. . 122
4.12 Hypertext adaptors socked to MacApp and ET++. ..................... 124
4.13 Class diagram representing dependency of TextView classes. .......... 126
4.14 Modified Node class diagram. ............................................. 128
Chapter 1

Introduction

1.1 Motivation

Software reusability is not new. It has been practised in various forms since the 1950s, mainly through the use of program libraries and common function libraries. Until recently, little has been done to extend the concept of reuse beyond this rather simple level. With significant technological advances in hardware technology, and other related fields, the software industry is being left behind. The growth of the software market and the recent realization of the importance of software in any product has initiated considerable research into ways of developing better software, at lower costs and with increased programmer productivity. Software reusability has been identified as a major component in bringing these improvements.

Much of the motivation behind this research comes from the importance of reuse, and its support by object-oriented technology. Object-oriented programming has been widely regarded as a suitable paradigm, promising improvements in quality, productivity, and maintainability. Reusability has been proposed as the main method by which these benefits can be attained [Meyer 87]. It is this promise of reuse, among its support of software engineering principles (e.g. modularity) [Meyer88], which attracted our attention.

Application frameworks like MacApp [Schmucker86] and ET++ [Weinand et al. 89] provide classes for creating applications for the Apple Macintosh and Sun Workstations respectively. These frameworks provide classes that handle all general aspects of an application such as the user-interface and basic file communications; only domain specific
behaviours need be added. These class libraries are based on similar design principles. A major aim of this project is to design "plug-compatible" components that can be used with either of the above mentioned frameworks or other similar frameworks.

The challenge comes in terms of understanding the functionalities offered by the frameworks and finding a common interface that domain specific classes can use. The strength of object-oriented technology lies in its ability to define structured libraries of classes and the ability to use inheritance to extend these classes to define new abstractions. The application frameworks provide class libraries and we wanted to explore how a new abstraction for a specific domain can be used with different frameworks.

1.2 Objectives

A specific application domain had to be selected to test the design approach. We have chosen 'Hypertext' as our application domain; the reasons for this choice are discussed in chapter 2. In other words, hypertext classes are to be designed for MacApp and ET++ frameworks.

It was necessary to understand MacApp and ET++ classes so that we could find the commonalities and behaviour of these libraries that allow the implementation of a portable "hypertext" class cluster. This objective can be achieved by:

- Understanding the working of MacApp and ET++.
- Identifying hypertext specific classes and objects.
- Generalizing and specializing the above class clusters.

A further objective to test the design, was to implement the class clusters on MacApp and ET++ application frameworks. The degree of reuse can be directly reflected by the ease of implementing the same class clusters across different platforms.

A final objective was to evaluate the success of the design approach, as implemented in the hypertext cluster, by adding new functionalities in the previously developed classes and by changing the requirements of the system. These methods should explore the reusability of hypertext classes. Suggestions should also be made to improve the reusability.
Chapter 2 surveys the state of research into reusability. It examines the motivations for reuse and sketches a brief description of a few reuse based projects. It introduces the basics of object-oriented technology and outlines its relevance to the topic of reusability. The chapter is not a tutorial in object-oriented concepts but attempts to define concepts within this thesis. It explores various class libraries and frameworks based on object-oriented paradigms. The problems related to these class libraries in providing reuse are discussed. The research goals of this thesis are presented in more detail. The choice of application domain is discussed. Then a literature survey on the currently existing hypertext systems is done. This is followed by the identification of components in a hypertext system. We conclude in chapter 2 that the object-oriented paradigm has the potential to provide an extensible, reusable software solution.

Chapter 3 presents requirement analysis and design of a hypertext system. The chapter starts with the requirements of a hypertext system. It also describes the main mechanisms used in ET++ and MacApp. While detailing these mechanisms, it identifies the commonalities available in these frameworks. The chapter does not explore the working of ET++ and MacApp in full detail, as that is beyond the reach of this work; it outlines important mechanisms specific to the needs of a hypertext program. Lastly, we identify the various classes and objects required in our hypertext system. The class clusters were designed to provide reusability.

Chapter 4 describes the implementation of classes identified in chapter 3 and evaluates the design. The chapter first implements the class cluster using the ET++ framework. Screen dumps of the ET++ based hypertext system are also shown. Next, the same classes are implemented on top of MacApp classes. During the MacApp implementation, the degree of generalization and ease of implementing the hypertext classes is outlined. Screen dumps of the MacApp implementation are also shown. The stability of class clusters is first tested by adding new functionalities in the system, and later tested by changing the requirements of the system. The chapter also presents directions for future work. We conclude by discussing the benefits of designing plug-compatible class clusters.

Chapter 5 is the concluding chapter of this thesis. It outlines the nature of reuse available in object-oriented concept based class libraries like ET++ and MacApp. It
mentions the pragmatism of the plug-compatible approach. The thesis closes with a reflection on object-oriented development, its close relationship with reuse, and techniques for improving reusability.
Chapter 2

Software Reusability and Hypertext - A Literature Survey

2.1 Overview

This chapter introduces the notion of reusability: an emerging area of interest within the software engineering community as it promises much in terms of addressing the main problems of the software industry today. Some of mammoth projects on reuse which are underway are discussed briefly.

The concept of reuse and the object-oriented paradigm go together. We have looked into reuse in action using object-oriented technology. The trade-off between building a domain-specific class library and reusability are discussed. The approaches adopted by framework builders are reviewed. The problems with reuse and our initial aims for this research are detailed. As a prototype study, 'hypertext' is chosen and a few of the more popularly known hypertext systems are addressed.

2.2 Reusability - Background

Reuse has always been a part of the problem solving process. As the solutions to different problems are found, these solutions are reused to solve new similar problems. The components of older solutions are adapted to solve a new problem. This is practised in all disciplines of science and engineering. But only certain components of a solution can be
successfully applied to new problems. With experience these components become established; they are standardized, generalized and accepted. For example, beams are standard components in civil engineering, nuts, bolts and gears are those in mechanical engineering, and resistors, capacitors and IC's in electronics engineering.

Since its development four decades ago, software engineering, like any other engineering discipline, has seen reuse in practice. But the degree of reuse has been very modest. Generic algorithms are the equivalent of mathematical solutions in other fields, best exemplified by work of Knuth [Wilhelm et al]. Subroutines, macros, functions and objects are examples of reusable software components. There are two factors that have prevented software from being widely reused:

- The nature of "Soft"ware is that it can be adapted, modified more easily than "Hard"ware. A hardware component cannot be adapted, modified or changed easily by a hardware designer, while a software designer makes changes in accordance with specific requirements. Thus the problem is evolving a set of standards for software construction.

- The second reason is identification of reusable elements. What components are available for reuse for a particular type of problem are not known. Meyer suggests a dictionary of such components [Meyer88] to solve this problem.

The above mentioned problems do not imply that software reusability is still in its infancy. The problem in Software engineering is not reuse, but systematic reuse. There have been substantial efforts made to solve problems via reuse, and some of this work is worth mentioning.

2.2.1 Birth of Software Reusability

Dough McIlroy is considered the pioneer of formal reuse [McIlroy76]. His seminar at the 1968 NATO Conference in Germisch, Germany, has become a seminal paper.

"Software components, to be widely applicable to different machines and users should be available in families arranged according to precision, robustness, generality and time space performance...Software production in the large would be enormously helped by the availability of spectra of high quality routines,
quite as mechanical designs are abetted by existence of families of structural shapes, screws or resistors.”

At about this time, key technologies like C and Unix and tools like Lex and YACC were being developed at AT&T Bell Labs, where McIlroy was working. The first case of formalized reuse at the organizational level was achieved by Lanergan and Poynton [Lanergan79]. They observed that 60% of all the business application designs and coded programs were redundant and could be standardized and reused. In 1980, Peter Freeman established the first university reuse research project at University of California, Irvine, with long term objectives [Freeman80]. The reward of the Freeman’s work came with Jim Neighbors’ dissertation, *Software construction using components* [Neighbors80]. Neighbors proved that software systems can be created by integrating reusable components. This gave a major boost to evolving software reuse practices. Similar kinds of achievements were obtained by Japanese companies. Hitachi pioneered systematic reuse in Japan. The project “Hitachi Software Factory” aimed to make a library of frequently used programming structures, giving rise to “reusable pattern modules”. Toshiba also formed a workbench called “Software Workbench”. The aim was not only to reuse standard code, but also standard designs, documentation, and test cases.

2.2.2 Reuse Programs

It was only in 1983 when the first workshop on reusability was organized, *Workshop on Reusability in Programming* [Workshop83]. This workshop gave a new direction to reusability and actually is considered as a turning point in the field. Subsequently, reusable products started gaining market attraction. A few of the most widely known reuse projects are:

1. **STARS**: The STARS (Software Technology for Adaptable, Reliable Systems) project was initiated by the U.S. Department of Defense in 1981. The aim of this project was to improve productivity while achieving greater system reliability and adaptability [Riddle et. al]. The focus of this program was mostly towards reuse technology and use of Ada. This program has undergone several stages since then, and during 1992 the focus was on Conceptual Framework for Reuse Processes
The idea is to make an integrated repository of all libraries developed under the STARS project.

2. ESF: The Eureka Software Factory (ESF) is the software factory of reusable components developed by European nations. There are 19 European countries participating in this 10 year program (established in 1986) and over 300 organizations are also involved. The aim of this consortium has been to create a market for environment components. The focus has been the development of a tailorable software environment supported by standard "plug-compatible" tools [Ohlsson90]. The US and Japanese approaches have focused on reuse from the application components point of view; in contrast, ESF has focused on the tools.

3. ESPRIT: The European Program for Research in Information Technology (ESPRIT) is a ten year program started in 1984. This program was divided into two phases, with Phase I results reported in 1989. In Phase I, the major emphasis was on Information Technology and there are two main outcomes of this phase, PCTE (Portable Common Tools and Environment) and Graspin, an analysis and design environment supporting structured development [ESPRIT89]. Both environments support reuse of tools through tailoring. The Phase II of ESPRIT is currently in progress and its emphasis is on REBOOT (REuse Based on Object-Oriented Techniques). The goal of this phase is to develop, test and disseminate an industrial environment to support software reuse by means of object-oriented technology.

2.2.3 Changing Definition of Reuse

The definition of reusability has evolved with time and better understanding of the concept. McIlroy defined reuse as the use of standard, off-the-shelf software components as building blocks in new larger systems. This definition is restrictive in the sense that it was aimed at reusing source code only. This black-box reuse of source code components originally mentioned in the definition was found to be inflexible and restrictive. In 1983, Peter Freeman gave a new twist to this definition:

"The use of program code alone has almost no value. It is entirely inconsistent to exhort developers to put more effort into the analysis and design activities and not attempt to reuse the information generated there." [Freeman83].
Freeman's definition of reuse made it more flexible and the concept broader. He emphasized that reuse of design and analysis are as important as source code to develop new software. Later, in 1988 Vic Basill advanced an even broader definition of reuse, the "use of everything associated with software project including knowledge" [Basill88]. This new perspective of using AI concepts of knowledge representation and knowledge acquisition for reuse, gave reusability a new vision. The inference of this new concept is that the reuse-based software development is continuous learning cycle guided by domain analysis. Software development is an ongoing process of continuous evolution that contrasts with the discrete step-by-step process represented by the more traditional waterfall model. It was realized that relatively new paradigms of object-oriented technology would handle the reusability much more efficiently than procedural paradigms.

2.3 Object-Oriented Paradigm

The main idea of the object-oriented paradigm is to move the focus of program design from algorithms to data structures. The data structures are represented in the form of objects. There are many definitions of an object: "An Object has state, behaviour, and identity; the structure and behaviour of similar objects are defined in their common class; the terms instance and object are interchangeable" [Booch91]. Objects are often related to each other in a particular way. Objects are arranged in a set, called a class, marked by common attributes or a common attribute. "A class is a set of objects that share common structure and a common behaviour" [Booch91].

Each object has a crisply defined boundary [Cox91]. But often objects are just like one another, with some slight modification or addition. So there is too much duplication if such objects are completely separate. This indicates that reuse of common attribute among nearly similar objects is quite natural in object-oriented technology. To separate the common property shared by a group of objects, generality is introduced. For example, while designing classes which represent the shapes of various geometrical figures; we can say that class Shape is an abstract class and implements some of the common properties which may exist in any kind of shape object (e.g. rectangle or circle). The subclasses of the Shape class can use generalized behaviour. The generalized behaviour is shared by a
group of objects with the help of inheritance.

class Shape
{
 public:
  
  ...
  //each subclass of this class should implement
  //its own draw method
  virtual void Draw();
  ...
  //shape class details the method for saving any shape
  void PrintOn();
  void ReadFrom();
  ...
 private:
  
  ...
  Point p;
}
class CircleShape : public Shape
{
 public:
  
  ...
  //circle specific draw method
  void Draw();
  ...
}
class OvalShape : public Shape
{
 public:
  
  ...
  //oval specific draw method
  void Draw();
  ...
}

Hence in an object-oriented environment, when writing a program, a programmer will
reuse code already developed by inheriting that code from more general objects or classes.
Thus code reusability is an inherent part of the object-oriented environment.

2.3.1 The Principles of Object Oriented Technology

There are four essential principles of the object-oriented paradigm relevant to the con­struction of object-oriented software. These principles are data encapsulation, data ab-
- **Data Abstraction:** The process of recognizing similarities between certain objects, while ignoring the details, is known as abstraction. In order to express the commonalities shared by identical objects, the object-oriented paradigm introduces the concept of class, as mentioned before.

The essence of data abstraction is that the user is presented with a higher level of abstraction over both the data and the algorithms required to manipulate the data. Thus, the user view is of operations which collectively define the behaviour of the abstraction, and is unaware of the internal details of implementation. The concept of abstraction supports reusability by allowing programmers to write software for generic purposes.

- **Data Encapsulation:** Data abstraction and encapsulation are complementary concepts: abstraction focuses upon the outside view of an object and encapsulation prevents clients from seeing the inside view, where the behaviour of the abstraction is implemented [Booch91]. The role of encapsulation is information hiding, where internal representation of an object is hidden from its clients, who may only access an object through its interface.

A complex system is broken into self-contained entities or modules or classes. A module represents a complete description of a part of the overall system structure, that is, it contains all the data structures and algorithms required to implement that part of the system.

The interface of a module normally consists of a number of operations which collectively define the behaviour of an object. The interface that is relevant to the internal details of a module is not available to the clients and is encapsulated. Thus the object-oriented paradigm supports modular design. The collection of such modules forms components. The user of these components only needs to understand the object interface and not the implementation, which consequently favors reusability.

- **Inheritance:** Classes provide a mechanism to declare the structure of a group of objects. With classes alone, all new classes must be created from scratch to
define the different functionalities of the whole system. This might be unnecessary as many classes will have similar behaviour. The object-oriented paradigm provides a tool in which classes may be categorized into a parent and child relationship, where a child inherits the behaviour of its parent class or classes. This is known as inheritance in the object world. Basically, inheritance defines a relationship among classes, wherein one class shares the structure or behaviour defined in one or more classes.

Inheritance supports reuse by providing a child class with the behaviour of the parent class. The commonalities shared by a set of classes are placed in an inheritance tree, where the root class represent the abstract behaviour shared by one or a group of branch classes. Classes that define only a signature and leave the implementation to be defined by other classes are termed abstract classes. Classes that define the implementation of a particular signature are termed concrete classes. Abstract classes define the data abstraction and concrete classes define the implementation of this abstraction.

Inheritance can be of two types: inheritance of implementation (or structure) and inheritance of specification (or behaviour) [Sakki89].

- Implementation hierarchies are based on deriving classes when it seems convenient to do so, usually in modeling implementation changes in a system. In such hierarchies a subclass may be derived from its parent class for the purpose of inheriting features and operations, that is to code reuse.

- Specification hierarchies base the arrangement of classes upon ensuring that a subclass is compatible with its base class, that is, the subclass IS-A base class. Such a scheme implies behavioural compatibility between the parent and child class. If the relationship is of type IS-A, then we should be looking for design reuse, not just code reuse [Coplien92].

• Polymorphism: Polymorphism is the ability of behaviour to have different interpretations in different contexts. This allows the derived classes to give their own meaning to redefine methods\(^1\) defined in the common base class. For example, an

\(^1\)A method in object-paradigm is analogous to function in procedural-paradigm.
abstract class *Collection* may define a method *print*, which does not carry out any work, but all the concrete classes like *Sets* and *Links*, derived from the collection class, provide the method *print* in their respective class interfaces.

Polymorphism can actually be introduced in two ways in an object-oriented environment [Blair et al.]:

1. *Through Sub-classing* — a method defined on a particular class is automatically defined on all its sub-classes.

2. *Through Overloading* — a method may be defined in independent parts of an inheritance hierarchy and hence to overload the meaning of the methods.

Polymorphism is one of the most important characteristics of the object-oriented paradigm and it shifts the focus of the designer and programmer towards the abstract behaviour of an object, rather than how this behaviour is implemented.

### 2.4 Object-Oriented Reuse

Object-oriented programming and reusability are often closely linked. One of the major factors for adopting object-oriented technology is "reuse". The reusable component in the object-oriented model is a class: an encapsulated, well-defined entity with distinct attributes and operations.

#### 2.4.1 Mechanisms of Reuse

The simplest kind of reuse is to use an existing component as it is. In object-oriented programming this is achieved by creating an instance of a class and accessing it by means of its interface. This form of reuse is called *compositional reuse* as it uses components in constructing software. This type of reuse is very limiting as it requires that the component be suitable exactly as it is, that is, the components are not easily modifiable or extendable. These type of components are useful for mathematical software systems.

When a component is not suitable in its present form or partially unsuitable, then it may be tailored to suit the needs of the application. This is achieved by *inheritance*.
A new component is created by augmenting an existing class, thus creating a new class without affecting existing code.

Thus reuse in object-oriented paradigm is facilitated in two ways: composition and inheritance. Inheritance can be used to create objects that extend or restrict existing objects and composition can be used to assemble composite objects. Inheritance also proposes problems for reuse in terms of exposure of variables which can violate encapsulation rules [Snyder 87]. Such problems must be considered when designing and later coding for reuse in the object environment.

It is clear that reuse is ‘natural’ in the object-oriented paradigm. Let us examine reuse in action using the object-oriented paradigm.

2.4.2 Examples of Reuse

The Smalltalk class hierarchy was the first widely known reusable class hierarchy based on the object paradigm. Other object-oriented class libraries, which provide reusable and extendable components are, GNU library of classes [Lea 88], Eiffel class library [Meyer 88a], the NIHCL library in C++ [Gorlen 87], and numerous other class libraries.

It is an accepted fact that the area in which the object-oriented paradigm has been most widely used to achieve reusability is the Graphical User Interface (GUI). There are two types of GUI toolkits. One type offers a number of classes implementing GUI interaction components; and the other provides an application framework.

1. Interaction-components toolkit: Interaction objects are the user interface components of windows, menus, scrollbar, and other control elements. The interaction objects of a toolkit usually conform to a certain user interface style guide. Examples of object-oriented toolkits offering interaction objects are Motif [Young90], OLIT (Open Look Interensic Toolkit) [Young et al. 92] and XView [Heller91].

2. Application-framework toolkits: In addition to the interaction objects found in other user interface toolkits, an application framework provides several classes for the development of other useful components, that is, the model and controllers of an application. In application framework, there are often additional classes to provide event-driven control flow mechanisms and common data structures (e.g
arrays, lists, trees, hash tables). MacApp [Wilson et al. 90], NextStep Application Kit [NeXt91], ET++ Application framework [Weinand et al. 89], and Borland C++/ObjectWindows [Borland91] are examples of application frameworks.

These toolkits provide a high degree of reusable and extendable components. Some of the toolkits and libraries are discussed in more detail in the next section.

2.5 Class Libraries

Component Libraries

There are a large number of component class libraries available which are useful building blocks for the construction of other programs. These “building blocks” are concrete classes and represent commonly required abstract data types (ADT’s). The component libraries package the simple classes like Date, Time, Collection, Fraction (rational arithmetics), Point, and Rectangle.

A lot of similar component libraries are available through shareware on the Internet and are also offered by the commercial market. One of the most successful component libraries is the NIHCL (National Institute of Health C++ Library) class library developed by Gorlen and his colleagues [Gorlen et al.] [Gorlen 87]. There are 60 classes in NIHCL and the total source code is approximately 275MB. This library is based on a single inheritance philosophy, where every class is inherited from one base class Object. An example of the Collection class tree structure of NIHCL is shown in figure 2.1.

The success of these class libraries is limited by two main factors:

- Companies prefer building their own in-house libraries for specific components rather than using other free and commercially available class libraries, and
- The component libraries may not be able to cater for domain-specific requirements of individual companies.

But the useful abstraction might be developed into reusable component classes to fulfill the demands, although such development will require time and effort.
Design Kits

A design kit (toolkit) is a set of reusable components that can be combined in the construction of applications within a given domain [Fischer & Lemke 88]. The components of design kits are specific to the domain of interest as they offer more in terms of reuse and suitability, for example, the toolkit approach to software construction for the user interface domain.

Some examples of toolkits are the Andrew Toolkit [Palay et al. 88], the X Toolkit [McCormack et al. 90] and InterViews [Linton et al. 88]. The Andrew and X-Toolkits were developed in an object-oriented style, whereas the InterViews toolkit was developed in C++. These design kits offer a selection of relevant components that can be manipulated and experimented with, in order to compose applications. As Linton [Linton et al. 88] has quoted:

"The programmer should be able to pick and choose from among the predefined components for the bulk of the interface, and the toolkit should make it easy to synthesize those components that are unique to the application"

The toolkits are more complex than the component libraries. They have a mix of concrete and partially implemented abstract classes. As discussed in the previous section, the toolkits are mostly used for developing GUI-based applications. The typical components of a toolkit are: Menus (pop-up, pull-down, scrolling etc), Buttons (action-buttons, radio-buttons etc), Scroll Bars, Dialog-Boxes, Alert-Boxes etc.
The user of such a toolkit library can create instances of the fully implemented concrete classes. If the user wants to tailor any class of toolkit, then he/she may create a specialized class derived from some base class available in the toolkit. For example, if the user wants to specialize the `Action-Button` class of the toolkit, then a specific class `MyAction-Button` may be derived: `class MyAction-Button : public Action-Button`. MyAction-Button class may use the publicly available attributes of Action-Button class and may also extend the class for specific application requirements.

### Frameworks

A step beyond the toolkits is the application framework. A framework "defines much of an application's standard user interface behaviour, and operating environment so that the programmer can concentrate on implementing the application specific parts" [Weinand et al. 89]. Examples of such frameworks are the Lisa Toolkit [Williams84], MacApp [Schmucker86] [Wilson et al. 90] and ET++ [Weinand et al. 88] [Weinand et al. 89].

The framework provides standard user interface operations and details all non-application mechanisms such as: creation and deletion of windows and their constituent parts, handling errors, file management, all input/output operations and memory management. It is a class hierarchy which the user extends and uses in the process of developing an application. To create an application, users are required to incorporate their application specific code into the framework structure. This is achieved by subclassing application classes of framework to program specific classes. In this way the framework will correctly display and manipulate application specific code.

Application frameworks provide a structured method of ensuring a standard "look-and-feel" for any application. The purpose is to remove the cumbersome work of creating new user interface code for every new application from the programmer's shoulders.

Application frameworks are powerful tools which do provide code reuse. Apart from that, frameworks may also be utilized for design reuse. By using the design of the application framework classes, users may get the idea of building their own domain-specific classes.

Application frameworks are available for all the popular platforms with their own look-and-feel since there are no standards for frameworks. In this research, we are interested
in ET++ (Sun Workstations) and MacApp (Macintosh) frameworks. These frameworks are based on the Smalltalk-80 Model-View-Controller (MVC) philosophy, and provides classes for windows, views, dialogs, and controls, and also includes classes representing commands, documents, and the whole application. We will discuss more about these classes with respect to ET++ and MacApp frameworks in the subsequent chapters.

### 2.6 The Problem Domain

Techniques for developing interactive interfaces have seen a major boost in recent years. There are three basic design principles which are used to develop interactive systems:

- **WYSIWYG** (What You See Is What You Get),
- the elimination of *modes* in the interface, and
- **direct manipulation.**

People at Xerox PARC have been working since 1972 on the Smalltalk system to achieve interactive interfaces using the above mentioned principles. The Smalltalk-80 programming environment [Goldberg84] introduced a number of important architectural ideas for constructing an interactive user interface. The Smalltalk-80 user interface itself is built around an architectural framework called Model-View-Controller (MVC), which is the inspiration and starting-point for most subsequent frameworks.

Several members of the team who were earlier working on the development of the Smalltalk language, joined Apple Macintosh personal computers. This new team at Apple adopted direct manipulation\(^2\) guidelines and, based on MVC experience with Smalltalk, developed the MacApp class library. Following the commercial success of Smalltalk-80 and MacApp, the ET++ class library emerged. “The ET++ class hierarchy and the ET++ programming environment are strongly influenced by the design of the Smalltalk-80 system” [Weinand et al. 89].

Since ET++ and MacApp frameworks are based on the MVC model the question arises:

\(^2\)Direct Manipulation requires user-driven sequencing of actions, which implies event-driven programs.
1. Can we develop applications which may be plugged into both frameworks?

2. Can we model the user requirements in a cluster of classes and use the “plug-in” technique to combine the cluster with a different framework?

This will prove that if the class libraries are based on same design principles then inter-library code and design reusability is possible. The degree of reusability may be measured with the minimum number of changes required to plug the cluster across the libraries.

Research Goal

The research goals of this thesis may be sub-divided into three sections:

- The study of how a “cluster” of interrelated classes might be constructed. A cluster is more than just one abstract type but is not as complex as a framework.

- The application domain for a cluster of classes may be:
  - classes for image processing?
  - classes for a mathematical model?
  - classes for a hypertext system?

- It is hoped that some approaches can be found to build clusters that can be combined and used with different frameworks. In our case, the frameworks are ET++ and MacApp. Thus, can we build application-specific plug-compatible components that can be ported on different platforms?

2.6.1 Problems with Code Reuse

As we have discussed in the previous sections, the object-oriented paradigm supports code reuse. But the question is: what is there to reuse? It is argued that the object-oriented paradigm has helped to evolve libraries which can be used to build software by providing standard components for reuse. Currently available class libraries can be divided into two categories:
1. Libraries with useful but with relatively trivial components. Examples of such component libraries are NIHCL (National Institute of Health C++ Library) and LEDA (A Library of Efficient Data Types and Algorithms). These libraries muster components like *Linked List, Date, Rational numbers etc*, see Section 2.5 for more details.

2. Libraries which model a program. Examples of these libraries are various *toolkits* and *frameworks*.

While such libraries do provide code reuse, the degree of reuse is hampered because:

- The classes in libraries are too extensively interconnected, so porting software across the libraries is hard, if not impossible.
- The framework libraries are huge, and understanding them takes considerable time before one can exploit these libraries.

The idea is to build more generic applications which can be combined with various libraries. The creation of a generic application is the aim of this project.

### 2.6.2 Reusable Clusters

A *cluster* may be defined as a group of classes that cooperate to provide some substantial services to clients. A cluster models the user's application domain, which can be successfully employed with different user interface domains.

As shown in figure 2.2, a reusable cluster of classes provides services to the clients, depending upon the underlying system. A strong interconnection exists within the cluster, but there is weak interaction between the cluster and outside world. This weak interaction will make the cluster more generic so that it can be ported to different frameworks.

### 2.6.3 Design to Plug Approach

To connect a cluster of interrelated classes with different frameworks, we hope to find the "adaptors" through which we can plug a cluster onto different frameworks. The cluster will depend upon the environment, e.g., for input/output.
Figure 2.2: A cluster of classes providing some service to the clients.

Figure 2.3: The Adaptor Approach.
Thus to provide the generality, there is a need to standardize the cluster for different frameworks so that the dependencies of a cluster can be funneled through well defined adaptors, see figure 2.3. For example, all frameworks have classes for I/O defined in their respective libraries. These I/O classes depend upon the specific platform. The MacApp I/O classes handles object save/load differently to the ET++ I/O classes. This indicates that depending upon the requirements of application domain, we may have to develop more general I/O classes that can be ported to both the frameworks.

2.7 The Application Domain

The choice of example domain for our purpose was dependent upon two main factors:

1. The application domain classes should have a high degree of inter-connectivity among themselves to efficiently implement the interface of the application, and

2. The domain classes should exploit different frameworks for maximum code reuse.

While considering the example domain, three applications were short listed:

- Classes for Modeling Mathematical functions
- Classes for Modeling Image Processing
- Classes for Modeling Hypertext System

The problem with mathematical classes is that they would be too simple. These classes would just represent some abstract data types, like matrix or multiple precision numbers, with very simple interfaces. Therefore, they would be easy to port across the platforms.

The problem with image processing application classes is that they require far too much domain-specific knowledge and would change the focus from software engineering to images.

The idea of developing a hypertext type system was found to be suitable for this type of research. The cluster of classes will model the hypertext style information editors and browsers.
2.7.1 Why Hypertext as a Cluster?

Hypertext was chosen as the application domain because:

1. There are no hypertext classes available in some of the most common frameworks like MacApp/MFC (MicroSoft Foundation Classes)/OWL (Object Works Library). Although our work is limited to MacApp and ET++, developing something which is commonly missing in all the frameworks may suggest some future work in the direction.

2. The ET++ framework does have a few classes for hypertext systems, but they are a bit obscure. Moreover the ET++ hypertext classes are strongly tied to the framework.

3. Hypertext provides a suitable scale for our cluster. We require that our cluster should have strong interconnection within its classes, while it should be loosely connected with the frameworks. Since the hypertext editors and browsers will use View classes of underlying frameworks to support various displays, we can study the level of code reuse in ET++ and MacApp. The data-model classes in our system will have to design their own common classes (although their versions exist in ET++ and MacApp), for example, Collection and I/O classes, so that they can be adopted with various frameworks. Hypertext is an appropriate application for such a model.

Thus hypertext is an ideal candidate for our research. In the next few sections, we will discuss some common features of a hypertext system and a few examples.

2.8 Hypertext - A Survey

Hypertext is text in non-sequential order. A hypertext application consists of fragments of text, graphic, audio, video and other forms of data. These fragments are called nodes. These nodes are connected with links. The connection of nodes and links forms a network. This network allows the text to be organized in such a way that one can jump around from node to node. In simple terms the hypertext can be defined as:

"non-sequential reading and writing- allowing authors to link information, create paths through a corpus of related material, annotate existing texts,"
create notes and point readers to either bibliographic data or the body of the referenced text" [Conklin87]

While the term hypertext was coined by Ted Nelson in the late 60's [Nelson67], the concept was discussed by Vannevar Bush in 1945. Bush imagined a "memex" device that can be used by a person to store all information about books, pictures, records, letters and so on. "A device in which an individual stores his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility. It is an enlarged intimate supplement to his memory "[Bush45].

After Bush’s visionary device, the first serious work occurred after 20 years. In 1968, Douglas C. Engelbart at Stanford Research Institute demonstrated the use of his "Aug­ment" system to create a hypertext document with a colleague 500 miles away. Engelbart also demonstrated two of his other inventions, they were: the mouse and the cord key set. Other people also contributed to the hypertext work. As mentioned above, Nelson came up with idea of hypertext. He thought about hypertext when trying to collect information on cards and arranging them in a non-sequential manner. As he mentioned in his book Literary machine:

"Every file card wanted to be many places at once, many needed to be pasted in the middle of several documents and separately reworked, but needed to stay connected between documents as well. All methods of paper were wholly inadequate and imposed connective restrictions that masked the true structure of the ideas" [Nelson81].

Nelson’s first design was basically word processing with inter comparison of alternative versions, with historical back-tracking capability. His second design was chunk-free hypertext. A user reads a chunk of text and then decides which chunk to read next. In his third design stage he combined the concepts of side-by-side editing and comparison of documents with his idea about non-sequential writing. Through the years Nelson has continued to work on his hypertext system. He came up with the name Xanadu for his developing hypertext system. A brief description of Xanadu is given in the next section.

---

3Ted Nelson is also the pioneer of the term HyperMedia.
2.9 The Milestones

A number of hypertext systems began after Engelbart's demonstration in 1968. All of the early work was done in large computer systems because microcomputers had not yet been invented. Some of the important milestones include the following:

ZOG

Work on ZOG began at Carnegie-Mellon University (CMU) in 1972. It was developed from a summer workshop for researchers in cognitive science. It allowed the participants to easily interact with each other's programs by providing a uniform menu-selection interface. The work on ZOG then stopped due to inadequate technology. But later in 1980, the work on ZOG was rekindled with a series of ZOG versions for PDP-10s, VAXs and an experimental multi-processor machine.

The ZOG was used for a computer-assisted management system for the United States navy's nuclear-powered aircraft carrier, the USS Carl Vinson [Akscyn84].

Intermedia

The Intermedia was first developed in Brown University. It was developed to produce non-linear textual materials. The Intermedia system is built on top of the 4.2 BSD UNIX operating system and runs on IBM RT/PC and Sun Workstations which support Sun's Network File System (NFS) [Meyrowitz86]. The system environment is described by Yankelovich [Yankelovich87].

NoteCards

NoteCards is probably the best known hypertext system developed at Xerox Parc. Its purpose is to collect, analyze, store and organize information. Moreover, it is used to construct arguments and models, and produce reports. It is written in Xerox Parc's lisp language and runs on Xerox Workstations.

The basic components of NoteCards are notecards, links, browser, and file-boxes. NoteCards provides the user with a "semantic network" of electronic notecards interconnected by typed links [Halasz87]. A notecard can contain text, structured drawing or bitmap images. The browser is a hypermap, showing the connections between various cards.
Knowledge Management Systems (KMS)

Knowledge Management Systems (KMS) evolved from the ZOG project. The purpose of KMS is to facilitate collaboration between members of an organization on a variety of applications. KMS supports organization wide collaboration for a broad range of applications, such as electronic publishing, on-line documentation, project management, software engineering and computer-aided instruction. KMS runs on Sun and Apollo Workstations and displays frames in less than half a second, on average [Akscyn88].

The basic unit of KMS is frame, which is a node. A frame can have text, graphic images or nothing in it. Each frame consists of a title, system frame name, the body of information, a list of links, annotations, and a command line. A link is used to connect the frames. The links may be of two types: tree links which lead to hierarchically related information and annotation links which lead to peripheral material.

Neptune

Neptune is a hypertext system for computer assisted software engineering (CASE). It was introduced in 1986 as a generic hypertext system to support the design of documentation for large scale software systems. It enables writers, designers, implementors, and evaluators to collaborate on the development of a system and its documentation.

The user interface for Neptune is written in Smalltalk. It is supported by a server, the Hypertext Abstract Machine (HAM) [Campbell88], which creates images for all of the transactions requested through the interface. The HAM facilitates the creation, storage, editing, and accessing of nodes and links.

Neptune provides several types of browsers, the primary ones are document, graph and node. A document browser provides a hierarchical listing of topics. The graph depicts nodes as labeled boxes and links as arcs. The node browser accesses individual nodes in the document [Delisle86].

Xanadu

Xanadu is a dream of Ted Nelson [Nelson74] and is still evolving. Its an “industrial strength” hypertext server which is designed to run on a local area network (LAN), controlled by a Sun Workstation. Any computer, such as the IBM-PC or Macintosh that can
communicate with a Sun, can be used to develop the applications.

Xanadu will support all kinds of nodes stored in digital format i.e. text, sound, video, and graphics. Multiple back-end servers can be connected to each other via a LAN connection, but to the end user it will look like one network. Links are bi-directional, i.e., a back tracking facility is there. In addition to a type name, a link may contain other information like the creator of the link, date of creation and comments.

HyperCard

More recently a number of hypertext systems have appeared on micro computers. One of the most widely known is HyperCard, which Apple computers distribute free with Macintosh computers. HyperCard is an authoring environment and information organizer. Bill Atkinson, the creator of HyperCard, has pieced together, adapted and expanded other Macintosh application programs like MacPaint to produce HyperCard.

The basic node object in HyperCard is the card, and a collection of cards is called a stack. The HyperCard function at three levels: browsing, authoring, and programming. At the browsing stage the author navigates through the stack and may create new cards. At the authoring level, an author uses a variety of powerful fields, buttons, and paint tools. These tools may be added by the user in a card to give necessary semantics to the card. More linkages are created at the programming level, where the author may use a simple, structured, object oriented programming language called HyperTalk.

HyperCard can import text and graphics and export text as well as launch other programs. Recently, many HyperCard applications are being developed for computer assisted learning. Jakob Nielsen in HyperCard based package, HyperTEXT '87-Trip report, develops a hypertext system to give a guided tour of hypertext [Nielsen88].

Guide

Guide is one the first hypertext systems for IBM PC's and is based on the idea of HyperCard. It is developed by OWL International. Guide is also available for the Macintosh. The most distinctive feature of Guide is it sees a document as a single, continuous scroll rather than a series of nodes or cards. Guide links document substructure together rather than in fragments, thus users have the option of scrolling through the documents in a
linear fashion [Brown87].

Guide has three linking structures, which are *buttons or hotspots*. Buttons are linked with each other and these buttons can be documented in text and graphics. These buttons may contain other buttons for more detailed information. There are certain quick *reference buttons*, which take the user to different parts of documents.

**HyperTIES**

HyperTIES has been under development at the University of Maryland, Florida, US, since 1983. The TIES in the HyperTIES stands for "The Interactive Encyclopedia System". Rather than using a pointing device, HyperTIES uses the arrow keys to move a light bar onto a highlighted object. It was developed as a browser for information in libraries and museums but has evolved into a fully interactive hypertext environment for instruction, problem solving, and self-help manuals [Schneiderman87].

HyperTIES originally ran on IBM PCs but it has recently been implemented on Sun Workstations. Earlier it was written in APL, recently rewritten in C. The editor of HyperTIES prompts the user for titles, brief definitions, more lengthy text and synonyms for article titles. The user marks important terms in the text and HyperTIES collects those terms. The HyperTIES takes cares of the synonyms of terms marked, indexes the articles and provides notetaking. It also tells the user about the links of different terms. HyperTIES also provides a very simple wordprocessor for creating the text.

2.10 **Components of Hypertext Systems**

The basic building blocks for the hypertext system are: nodes, links, buttons, and editors/browsers. In this section we will discuss the properties of these elements.

**Nodes**

As discussed before, hypertext is a network of nodes; a node is a collection of data related to a specific topic. A node usually is limited to what can be presented on a single screen. However, a node can be scrollable or fixed. When we think about a node it might be of type:
Figure 2.4: A Hypertext Node linked with another node with help of marks.

1. Text

2. Graphics

3. Video

4. Audio

All the above are different ways in which information may be represented in a node. A node contains embedded information about other nodes. To move from one node to other, links are used. The links are generally represented with the help of marks. See figure 2.4. The behaviour of these nodes is as follows:

**Text Nodes:** A text node can be of scrollable or fixed type. The systems which allow the user to scroll through a long article are called *article-based systems*. Systems that offer only fixed screens are termed *card-based systems*. Guide is an article-based system while HyperCard is a card-based system. But in HyperCard we can have a scrollable field inside a card. This scrollable field in HyperCard is one of the many tools provided.

**Graphics Nodes:** Graphics nodes are the images that are displayed inside a node. These images can be of type character or bit-mapped. Apple’s MacDraw is a structured graphics editor, similarly Intermedia’s InterDraw is also a graphics utility.
Intermedia’s InterPix is another utility program that displays bitmap images entered into the system using a digitizing scanner. These images can be cropped, copied, and pasted into InterDraw documents. [Scully, 1988].

**Video Nodes:** In a Video node the hypertext program launches the video tape or videodisc segment. The control of such applications is through buttons which are references to certain section of video tapes or discs. A good example of video card is *NoteCard* hypertext system. In NoteCard, the video cards control a videodisc player that plays a designated video segment on a separate screen when the card is displayed [Halasz, 1987].

**Audio Nodes:** Like video nodes, audio nodes are also a typical hypertext characteristic. Musical Instrument Digital Interface (MIDI), is a system that allows the capture of a musical performance in digital form in computer data files. NoteCard do have the facility of audio nodes. There is a good program called WinSong for IBM PCs and compatible, which runs under Microsoft Windows. This program will record the performance of a musical instrument that is connected to the PC through a MIDI interface. Besides recording a musical performance in digital form, WinSong can translate the data to conventional musical notation. Editing musical notation is also possible.

**Links**

Links are used to connect two nodes. Links form a network between different nodes. Links may be embedded and highlighted or may be outside a node, and connect it in some way to another node. A link might be a file name or a node name. Links enable users to determine the sequence in which the information is represented, and allow the user to move around the hypertext. Links are activated by using the mouse or keyboard to point at a *hot spot* on the screen. By clicking the mouse, the node to which the link is pointing is displayed on the screen.

**Paths:** Hypertext links also create *paths* or *routes*, which are of pre-defined type. The hypertext user who does not want to make their own way through the system may use this default option. When developing a hypertext application the information in nodes is also presented in *guided tour* fashion.
Network Of Idea

When many nodes or cards are linked together in a consistent way, they form a network of ideas. The network, as mentioned above, is basically formed by links. The network provides the semantics to a hypertext system. The network is formed by associative links, i.e., one node is linked to another in an organizational way. The network of hypertext ideas may be constructed by the designer or the user to resemble the subject matter structure or the semantic network of the user [Jonassen 89].

User Interface and Interactivity

The user interface and interactivity of a hypertext are of foremost importance. The main difference between reading a book and the hypertext system is enhanced flexibility provided by the hypertext. By using good screen design principles, the user and the screen become more interactive. Some of the screen design principles are:

- Group all the related ideas.
- Use Graphics to stress the important ideas.
- All links and buttons should be self explanatory.
- At the end of each node, the user must choose where to go next.
- Keep the interface as simple as possible.

Database

If we look from the database point of view, nodes or cards are the repository of information. Hypertext facilitates the searching and accessing of information. This database depends upon the type of node, it can store text or graphics or both. The multi-user access is the other classic example of hypertext repository, where users can contribute to the database. *Wide World Web* is a good example of such a repository.

According to Conklin [Conklin87], there are three components of a hypertext system:

1. A database of text,
2. A semantic net which connects the text components, and
3. Tools for creating and browsing this combination of text and the semantic net.

The hypertext is text which is a combination of database, semantic network and interface. If the word “text” in the definition of hypertext is replaced by “music” or “film”, hypermusic and hyperfilm are types of hypercommunication or hypermedia [Yankelovich et al., 85].

2.11 Summary

This chapter has surveyed the current state of the art in reusability in software technology and presented a discussion of the motivation behind, and the key issues involved in the employment of a reuse strategy.

Reuse of design and code is becoming more common and the object-oriented paradigm provides an ideal environment for reuse. The object model’s principles of data abstraction, and inheritance strongly support reuse. Currently, there are many class libraries being developed to provide code reuse. These libraries are: component libraries, toolkit libraries and frameworks. The problems with these libraries is that the classes are too interconnected and the size of the libraries is too big; thus code reuse is hampered.

To improve reuse, we want to study how a group of interrelated classes may be used to represent a user’s application domain on different frameworks. The design to plug approach seems to be the best way of achieving such a goal, where adaptors are used to plug-in the cluster of classes with different frameworks.

Hypertext is chosen as the prototype domain for our application because it was found that hypertext editors and browsers construction will exploit the code reuse from frameworks, and will also model a suitable cluster of tightly interconnected classes. A survey of different well known Hypertext systems is done in brief, followed by different components identification required to construct a hypertext system.
Chapter 3

Hypertext System: Analysis and Design

3.1 Hypertext Requirements

The first step in developing software is outlining the requirements of the system. The requirements specification describes what the software can do and what it cannot do [Rebecca et al.]. In this section, we describe the requirements of our hypertext system.

Hypertext is a network of nodes; a "node" is a collection of data organized around a specific topic. In a network, each node is linked to some other node. The "links" associated with a node are represented by "marks". A mark in a text node might be a highlighted word or phrase; a mark in a picture node would be a highlighted region. So, hypertext data basically consists of:

- Nodes,
- Links, and
- Marks

A node may be a textual, graphical, sound, or video node, as discussed in the previous chapter. In our case, the requirement is a hypertext system which supports textual and graphical nodes.

We have chosen to represent one node per file. Thus, a file name corresponds to a node name. When we say that two nodes are linked with each other, we basically mean...
that two files have information about each other. This information is typically the name of a file. Thus, a file constitutes data about a topic and the names of files to which it is linked, see figure 3.1. The link between two nodes will be bi-directional because both the files should know about this link between them.

A *mark* is a visual cue in a node that alerts a user that a link exists. In other words, a mark is a visual representation of a link in a node. A hypertext system must allow a user to mark or highlight a text or graphical region in a node. The marks act as “active areas” or “hotspots”. By pressing a mouse button on these active areas, the user can cause the system to activate a link to display a different node, see figure 3.2.

The hypertext system has to be designed for two types of user community: *authors* and *readers*. Authors need *editors* to construct a useful knowledge network, while the readers use *browsers* to acquire the embedded knowledge.

1. **Editors**: Used by a person, who is experienced and knowledgeable, to build an information network.

2. **Browsers**: Used by end-users, naives, and computerphobics to learn about a particular subject by browsing through an information network.
EDITORS

An editor is the part of a hypertext system that enables a user to create a node and link it into the network. The information of a node is displayed on an editor. Thus, we may say that an editor is a window which displays the data related to a specific node. Since the data of a node may be of different types, in our case text and graphical, we need different types of editors. Our requirement is to build a text and a picture editor. Authors will need these editors to write information specific to a node, while for readers it will act as an “information presentation” system. The editors should be scrollable because we want to display all the data related with a node on one screen rather than on a series of window screens.

BROWSER

A hypertext browser is a program that displays a list of nodes and links available with each node. Like an editor, a browser will also display a window with a list of nodes and links. Our requirement is that a browser should display a list of nodes, and by mouse action on any node, links available to that node should be displayed on a window. Readers will use browsers to navigate through a hypertext system.

DIALOGS

To facilitate interaction between users and a hypertext program, there must be dialog boxes. For example, when a user wants to specify the name of a node to which she wants
to link another node, there must be a dialog box which provides such feedback to the program.

**ORGANIZING NODES (FILES)**

All the nodes should be stored in a directory; text nodes in `textdir` and picture nodes in `picdir`. This will give us a simple database of nodes. The other benefit of a directory-based collection of nodes is that we can design the system in a way that our links are looking at particular paths for available nodes. Thus saving and loading of nodes is done in their respective directories, see figure 3.3.

**EDITING SUPPORT**

There should be the facility for “do-undo”. In textual node, it would be doing and undoing typing while in pictorial node it would be typically doing and undoing of a graphical region. This implies that we will need certain handlers to perform editing functions.

**LINKS**

A user should be able to migrate from one node to another, either by clicking the mouse on the marked region, or through a menu which consists of a list of available links.
REQUIREMENTS-SUMMARY

In conclusion, the system must be able to:

- enter and edit text, i.e., a text editor,
- support graphical images (for Sun Workstations and Macintoshes), i.e. a picture editor,
- allow marking of a text and graphical image,
- save and retrieve the documents,
- provide dialog boxes for interaction,
- show a browser for the different nodes available, and links attached with each node
- insert links in a node to other nodes for reference purposes,
- show a menu of links available with each node, and
- support display of other nodes from a node with the help of marks and links.

3.2 ET++ and MacApp: Application Frameworks

Since we are building our tool using ET++ and MacApp, it is necessary to understand the basic class structure of these libraries. We will outline a description of some of the useful classes of these frameworks. Before this, it is important to point that the reader must be familiar with the C++ language and its constructs. We suggest Lippman’s [Lippman91] and Stroustrup’s [Stroustrup91] books to understand C++ language. Also, the object-oriented notation used in this research is taken from [Booch91].

Figure 3.4 shows some of the most important abstract and concrete classes in MacApp, while figure 3.5 shows the inheritance relationship among ET++ classes. These class libraries are similar in the way that they are derived from one common root class that contains a considerable amount of functionality. The notion of defining a single hierarchy containing all of the classes is often referred to as monolithic class hierarchy. There have been many criticisms of developing monolithic libraries in C++ because they impose
Figure 3.4: MacApp Class Library - Some of the popular classes.
Figure 3.5: ET++ Class Library - Some of the popular classes.
considerable overhead in terms of memory load, compilation and link time, run time, code size, and hidden dependencies [Coggins90], [Dearl90]. Still, such libraries are popular in the software market because of their ease of use and degree of reusability.

The base class for MacApp is TObject and that for ET++ is object. By convention, all MacApp classes are named T<something>; no such convention exists in ET++. As we can see, these libraries are fairly large, and one of the most difficult aspects of these libraries is understanding what mechanisms they embody.

3.2.1 The Command-handler classes

The framework libraries model interactive programs. The model envisages many different “objects” present at run-time. Each such object owns some part of the program’s data, and handles some user-entered commands, such as menu selections, keyed input, and mouse-based interaction. These objects are called “command-handlers”\(^1\). The framework libraries are built with the idea of having a set of standard command-handler objects. Some of the standard command-handler classes in any framework are:

1. **Document:** A document command-handler owns the data of a program, and looks after any changes to that data. It arranges the transfer of data between disk files and main memory.

2. **Application:** An application command-handler is responsible for overall organization of the objects that make up a running program. It performs work involved in translating low-level operating system events into “commands” that could be sent to other objects.

3. **View:** A view command-handler provides coordinate framework for the display of data, methods for drawing data, and methods for responding to mouse-based data selection and editing the screen.

The principal command-handler objects follow a typical pattern of interaction. For example, if an application object finds that it is dealing with a “Quit” command entered by a user, then it will remember to warn any document object present. The document

\(^1\)The term event-handler is often used rather than command-handler.
object will check the status of its data structure. If its data structure has not been saved since the last change, the document will ask the user whether the data should be saved. If necessary, the data is written to a file, and the data structure window and document objects will all be deleted. In a typical framework, there are hundreds of such patterns of interactions associated with standard user-commands and run-time events. The various command-handler classes are consequently quite strongly coupled.

3.2.2 Building a program on framework library

To develop a program using a framework library, specific subclasses have to be created for each of the major command-handler classes. The programmer has to define a principal command handler as a specialized class of the framework’s Application, Document, and View classes. For example, figure 3.6 shows some of the “command-handler” objects that will be present during the execution of the ET++ based “drawing editor”. At run-time, there will be a single instance of DrawApplication object, one or more DrawDocument objects, and for each DrawDocument there will be a PaletteView object and a DrawView object.

There are many other objects. A DrawDocument object will have a Window object that it uses to frame its two views. A Window object may have ScrollBar objects. Finally, there will be program specific objects, such as DrawStructure object.

The main command-handler classes of “draw editor” program are specialized classes of a framework’s following classes:

class DrawApplication : public Application {
public:
    // Replace some virtual functions
    ...
}
class DrawDocument : public Document {
public:
    // Replace a few virtual functions
    ...
    // Add some specific functions to manipulate
    // draw structure
private:
    DrawStructure* fStructure; // Add Some data fields
Figure 3.6: An executing program and some of the "command-handler" objects that will be present.
The command-handler classes have the responsibility of invoking appropriate command objects in response to user actions. For example, to save or load a data structure, a mouse-action, a menu command or responding to an event. ET++ and MacApp have predefined mechanisms for such actions. These mechanisms form the core concepts of a framework.

Save and Load Mechanism

It is often desirable to save and load an application program, e.g., user of our drawing-editor wants to save some drawing for later use. A drawing editor should provide capabilities such as saving and loading. To achieve this, MacApp and ET++ classes have special pre-defined functions which must be overridden to activate the mechanism.

The command-handler class Document is used to save the state of an application. A document object is typically created whenever an application is launched or whenever a user selects New or Open menu items in the standard File menu. Each of these actions invokes class application method DoMakeDocument. A domain-specific implementation of this method must be provided in the subclass of Application, i.e., DrawApplication. This method creates and initializes a document object. When a user sends the com-
mand Open for opening a document, the application method OpenOld is called. This method calls ReadFromFile defined in the document class, see figure 3.7. Then method ReadFromFile invokes DoRead function which must be defined in the subclass of document, i.e., DrawDocument. The method DoRead is specific to an application domain; an effective implementation of DoRead could consist simply of a call asking the new DrawStructure object (which is created with the new DrawDocument object) to read its own data.

**View Mechanism**

A new document object created by a user action Open will have to create those objects that are used to display the data. These objects are instances of specialized View classes created for the program along with things like a Window to frame these views and additional objects such as Scrollers and Scrollbars. Since the display structures are essentially program specific, the programmer of class DrawDocument must override the method DoMakeViews of MacApp or DoMakeContent of ET++.

When an application object receives a call to open a new window, it invokes the DoMakeViews or DoMakeContent of the underlying framework. These methods, overridden in the programmers document class, carry out the specific task of creating new windows.
depending upon program specifications. For example, when a new document is created on MacApp, a call is made by an application object to the window method `Update`, which in turn invokes the window method `DrawContents`. The `DrawContents` calls method `Draw` for the window itself and all its subviews. The `Draw` method must be redefined for each domain-specific view. The class `DrawView` and `PaletteView` of our drawing-editor program have the method `Draw` defined in their implementation, see figure 3.8. We will look into the working of the view mechanism in more detail as we build our hypertext editors and browser.

**Mouse Command Mechanism**

In a program like “drawing-editor”, there is considerable amount of interaction between the user and the program. Once the windows and views are displayed on the screen, the program waits for the next action from the user. The action might be picking a tool from a palette box, clicking, dragging, and sketching. But the first work of an application is to determine the type of event related to a window. An event could be associated with menu-selection, or the user trying to bring a window in front, or the user trying to do something on an active window. If the `Application` object can determine the event relates to an active window, it sends a message `HandleMouseDown` to that `Window` object.

The `Window` object must determine whether the work required is its own responsibility
or whether the work can be passed to its subviews. Responsibilities of a window object are: grow a window, close a window, zoom a window, and drag a window. If the work required is none of the above, then the subviews actually contain the point pertaining to the cursor position, and the HandleMouseDown message is passed to these views. The response to a view object is specific to the application-domain. If it was recognized that the mouse click is in PaletteView, then class PaletteView must have defined DoMouseCommand method in its implementation, see figure 3.9. The DoMouseCommand in a PaletteView will typically implement the selection of a palette item from the available palette menu.

Menu Command Mechanism

The low-level event data are provided by all operating systems to characterize a keystroke or a menu selection operation carried by a user. But the event data cannot identify the particular object that has to deal with these kinds of user requests. Frameworks provide an elaborate mechanism to relate command-handlers and commands. All the command-handlers in ET++ and MacApp, like Application, Documents, Views, and Windows, are linked together. At any particular time, one particular command-object will be identified as the primary target for incoming menu or keystroke commands. This target object is identified by some global pointer or by some pointer data member in the Application object.

A command chain is built by frameworks whenever such a global pointer is initialized, \texttt{g<Target>}. In this command chain, every subview (scrollers, palette etc.) is made to point to its superview (windows), every window is made to point to its enclosing document, and every document is made to point to its application object. The chain does not start with the application object, but the first go is given to the subviews. If the subviews cannot handle the command, then it is passed to their superviews and so on, see figure 3.10. The global pointer invokes method DoMenuCommand in the subview. The developer must provide DoMenuCommand method to supply domain-specific behaviour. If the views cannot handle the user request command, the application object is given the command to handle. For example, commands like \texttt{File/Open}, \texttt{File/Close}, \texttt{File/Quit} etc are provided in the File menu of window and must be handled by the application object while user commands like \texttt{View/Refresh}, \texttt{View/Mark} must be handled by the window object.
3.2.3 MacApp

So far we have discussed general mechanisms used in ET++ and MacApp to deal with user actions. In this and next section we will briefly outline some information about the principal classes in MacApp and ET++ respectively.

There are over two hundred classes in MacApp [MacAppRef], these include:

- **Classes representing simple “Abstract Data Types”**: These include class VRect, VPoint, CRect, CPoint, CRGBColor, and class CString. Class VRect and VPoint handles 32-bit coordinate system, while class CRect and CPoint handles a 16-bit coordinate system. Class CRGBColor supports encoded red, green, and blue color data and class CString provides overloaded operators to enhance string handling capabilities.

- **Class TObject**: This is the base class for most of the other classes in the MacApp system. This class defines the dependency handling mechanism, support for object I/O, the cloning mechanism, and a scheme for run-time class identification.

- **Container Classes**: MacApp supplies container classes such as TList and TSortedList. There are other specialized subclasses such as, TClassByName, TClassbyID etc. At the root, all list classes are built using TDynamicArray class, which can be used
Figure 3.11: The “Command Handler” classes from MacApp library.

directly.

- **Iterator classes:** There are about twenty iterator classes, derived from abstract class `CIterator`. These classes provide the capability of iterating through a list of objects, for example, the class `CDocumentIterator` allows moving along the list of documents that belongs to a specific `Application` class.

- **Command Classes:** MacApp provides an abstract `TCommand` class to assist in the implementation of “do” - “undo” editing operations. There are specialized classes for tracking mouse, like `TTracker`, to assist mouse based editing. Apart from this, there are other concrete classes to support close, quit, cut, paste etc, for example, `TQuitCommand` and `TCloseWindowCommand` classes.

- **Command Handler Classes:** The general working of these classes were discussed in previous section. The hierarchy of specialized `TCommand Handler` subclasses is shown in 3.11.

- **Object I/O:** Class `TStream` provide support for object i/o in the MacApp hierarchy.
• "Toolbox" classes: There are a number of classes that simply encapsulate Macintosh Toolbox data structures, and provide some limited set of member functions that facilitate the use of structures.

### 3.2.4 ET++

ET++ class library consists of over five hundred classes [Weinand et al. 89]. The total code is around 50,000 lines and is written in C++. The ET++ library is far more extensive than MacApp library. It is designed so that the classes are pretty much independent of the underlying platform. Operating system dependencies, and platform dependencies are handled through interface classes and low-level implementation classes. Programmers working with ET++ rarely need to be concerned with the implementation of these low-level internal classes. Classes in the main ET++ framework library and its various subclasses include:

- **Classes representing simple "Abstract Data Type"**: As in MacApp, there are classes such as **Point** and **Rectangle** that implement simple abstraction for points and rectangles. Then there are classes for color support and bitmap. Class **RGBColor** and class **Bitmap** are derived from class **Ink**. Instances of class **Ink** can be created. ET++ has class **String** for string handling. Apart from this, ET++ has class **Date** and **Time**.

- **Class Object**: The base class of the ET++ hierarchy is class **Object**. This class defines mechanisms for run-time class identification, flag manipulation, object i/o, dependencies, debugging and cloning. There are a number of classes which are derived from class **Object**, see figure 3.5.
Figure 3.13: The "Command Handler" classes from ET++ library.

- **Stream Classes**: ET++ has classes `IStream` and `OStream` defined for object i/o. These classes possess the functionality of normal "C++" `istream` and `ostream` classes. But they are superior to the MacApp `TStream` class as they provide more functionality.

- **Container Classes**: ET++ has a set of container classes, see figure 3.12. These classes support the collection of objects in a list and sorted lists.

- **Command Classes**: ET++'s class `Command` combines MacApp's class `Command` and class `Tracker`. These classes provide capabilities like: `Do`, `UnDo`, `ReDo`, `TrackMouse`, and `TrackMouseConstrain`.

- **Command Handler classes**: As in MacApp, all ET++'s command-handler classes are derived from class `EvtHandler`, see figure 3.13. The class `EvtHandler` defines the basic interface for command handling. There are two subclasses of `EvtHandler` class - `VObject` and `Manager`. Unlike MacApp, the `View` class in ET++ is a specialized class of `VObject`, thus it implements one extra layer of abstraction.
3.3 Hypertext: Analysis and Design

So far, we have outlined the requirements of our hypertext system and with respect to that we have discussed some of the most common classes and mechanisms in ET++ and MacApp. In this section, we will structure the model of our hypertext system into classes and objects.

There are three main aspects to the design of a framework program. These are the data-model, the display structure, and the overall organization of program. The data-model is necessarily specific to an application; in our case the data-model will specify the behaviour of “nodes”, “links” and “marks”. The display structure defines some sets of views; there will be different view structures for hypertext “editor” and “browser”. Finally, there are components that organize the program, for example “application” and “document” objects.

The next three sections examine each of the three main parts. This will give us an idea of the classes and objects. In section 3.4 we will discuss the interaction among objects that might exist during the execution of an hypertext application.

3.3.1 A Data Model for Hypertext

From the requirements, we know that a hypertext program consists of nodes, which embody information on a specific topic. Each node owns links to other nodes and marks as highlighted areas. To put in object-oriented terms, the properties of any hypertext program include a collection of nodes, a set of links associated with a node, and a set of marks associated with a node. This captures the data-model of a hypertext program. Thus, we can identify the main data-model classes. In doing so, we will also point out the key abstraction in these classes.

Class HTNode

Nodes are good candidates for being objects as they represent data elements in a hypertext system. So, it is possible to proceed with a class HTNode. Each node will own some data. This data may be text data or picture data. The class HTNode also has some responsibilities; at the initial stage we can outline a few of them as:
1. A node should be able to draw.

2. A node should be able to save and delete its contents.

3. A node should be able to link itself to other nodes.

4. A node should be able to highlight its contents.

5. A node should provide editing facilities that allow an “editor” to change its data.

Figure 3.14 shows the data owned by class HTNode and various methods defined in the class. A node object owns mark and link objects and it constructs a list of nodes; the role of such a list becomes clear in the next paragraph. The mark and link objects will be supplied to this class by the calling class, that is, the class which will construct the objects of class node must pass mark and link objects as arguments. The constructor of this class will copy passed objects into its own data fields. With the help of these objects, the class will carry out the work of linking and marking.

The constructor will also construct a list of nodes. The role of such a list is when the program begins, the names of files in text and picture directories should be stored in this list. As per system requirements, a node is basically represented as a file, so when a user deletes or constructs a file, appropriate deletion/addition action will be taken by the node object. This class will provide methods for adding, deleting, and finding nodes from the
list (i.e., files in any specific directory). The responsibilities of saving the contents of a node and loading a node are performed by methods PrintOn and ReadFrom respectively.

Class HTextNode and HTPicNode

The next step is to identify the key abstraction in the HTNode class. The node can be of two types; a text or picture node. A text-node object will represent data for a text based hypertext editor and a picture-node will represent data for a picture based editor. Though class HTNode embodies generic behaviour of any type of node, the requirements of graph and text based nodes are different. There is a need for subsequent specialization of class HTNode into class HTextNode and class HTPicNode, see figure 3.15.

Depending upon the type of editor a user needs, a text or picture node object would be constructed. They will inherit most of the functionality from the class HTNode but they will also have additional or modified functionalities. For example, the saving and loading of text and graph nodes are different. The ReadFrom and PrintOn method in class HTNode will become virtual and each of these classes will define their own ReadFrom
and *PrintOn* methods, refer figure 3.16.

The responsibilities of **HTNode** class at this stage are still not complete. This is due to the fact that interaction among application, view, and data objects will add more work to the HTNode object. For example, the mouse-click handling will have an impact on HTNode objects. We will add more methods in this class at a later stage to deal with such situations.

**Class HTLink and HTLinkManager**

The hypertext program needs to make links between two nodes. A node is typically a file: text or picture file. This implies that when we make links between two nodes, we are actually linking two files. Each node object knows about all the links to which it is attached. A node should be able to create new links or delete old links. Such operations may be defined in the node class, but since "links" represent completely different abstraction, we have chosen to define the **HTLink** class, see figure 3.17. The other main reason for defining **HTLink** class is from a maintenance point of view. In the future, it would be easier to modify the behaviour of "links" without touching the "node" classes. The class **HTLink** will have the responsibility of:

- Adding links for a specific node.
- Deleting links for a specific node.
- Saving the links in a node specific file.
- Loading the links of a specified node.

The interconnection between class **HTNode** and **HTLink** is strong. An instance of the **HTNode** class owns an **HTLink** object, and each of the node objects owns a list of links, see figure 3.18. For information about available links, the **HTNode** object relies on the **HTLink** object. How the objects actually communicate will be described in the following sections.

Apart from the mentioned above responsibilities, the **HTLink** object must also provide the capability of interaction between data model, application and document objects. This is because if a user request the opening of a node which is linked with the current node, then the **HTView** (or **HTPicView**) object will pass the message to the **HTNode** object.
Figure 3.17: A cluster of link classes - HTLink and HTLinkManager classes.

Figure 3.18: Interaction between node and link objects.
to carry out the work. The HTNode object will consequently send the message to HTLink object. The HTLink object must handle the operation of opening the desired node. To handle open/close operations of documents, it is necessary that HTLink class must have HTDocument and Application objects (in frameworks, these classes start the chain to handle open/close mechanisms, see section 3.2.2 for details). We can have data fields for document and application objects in the HTLink class. But an HTLink object should not be responsible for such an operation, as this is not a legitimate role of HTLink class. Instead, a "LinkManager" object should manage such an operation. The HTLinkManager class will have method ShowLink which will take application and document objects as arguments and will open a document on the user's request, see figure 3.17.

Class HTMark

The hypertext program must allow the user to mark a text word or graphical region. An HTMark class is a suitable abstraction. Like HTNode and HTLink classes, a mark object will be created by the document object. The mark can be of two types; depending upon the hypertext text or graph editor, the document object should be able to create different types of marks for different types of editors. This means that there will be specialized classes of HTMark, which will provide unique services (i.e. class HTextMark and HTPicMark).

In figure 3.19, we have outlined some of the main responsibilities of the cluster of mark classes. The HTMark class has default constructors and owns a list of marks. Both HTextMark and HTPicMark class have a method MakeMark defined in their respective implementations. The method MakeMark will actually either highlight a word (in the text editor) or mark a region (in the picture editor).

3.3.2 Views for Hypertext

The view classes for the hypertext system are: displays for two kinds of editors, display of a browser, display of dialog-boxes for interaction with users, and display of a palette for picture editor. The data model classes are reusable as they are least dependent upon any framework. But the domain-specific views will strongly depend upon frameworks. We have to find the commonalities so that the classes may be ported on different platforms.
Figure 3.19: A cluster of mark classes - HTMark, HTextMark and HTPicMark classes.

At this stage, we will design program specific views for both ET++ and MacApp classes.

Class HTextView

The HTextView class will construct a text editor. ET++ and MacApp frameworks have class TextView defined in their respective cluster of View classes. We can thereby reuse the TextView class of frameworks and tailor it to the hypertext editor. In figure 3.20, the class hierarchy and responsibilities of HTextView are illustrated.

Figure 3.20: HTextView class.
Figure 3.21: The view components of a picture based editor.

An HTextView object will be created by the document object. As discussed in section 3.2.2, the view mechanism of framework libraries is quite complicated. When a document object creates a view object, some of the methods of view classes are invoked by the underlying framework. Such methods can be overridden in specialized view classes. For example, the Draw method in the view classes. In our case, we want a simple text editor with a menu-bar. Our document object will construct the views on the user’s request (discussed in next section), but “mouse-handling” methods must be specified in this class as a view object has to handle the user action within its physical boundary.

The main work done by this class is as follows: construct different application-specific menus, do some action when a user sends some request through menus, and handle the mouse click when a user clicks in the main view of the text editor.

HTPicView and HTPaletteView classes

Unlike TextView classes available in the frameworks, there are no PictureView classes currently available. So, we have to build our own HTPicView class, derived from the View class of frameworks. The HTPicView will help to construct a picture editor. A picture editor will have three main components: a “main view” to show the graphical objects, a “palette view” to choose some graphical objects and a “menu bar” for handling the user’s menu based commands, see figure 3.21.

The responsibilities of the HTPicView class is similar to the HTextView class. But since a picture editor comes with a palette, a PaletteView class is also necessary. This class will again be derived from the View class of the underlying framework. The actions to be taken when a user clicks mouse on a palette view or the main view of a picture
editor are different, so both these classes have their own *MouseDownCommand* and *Draw* methods, see figure 3.22.

**HTBrowserView class**

Authors and readers of a hypertext system need browsers to investigate various nodes and links available in the system. An author will need to be able to review different nodes and links in a partially assembled hypertext collection, while a reader is interested in navigating through the hypertext system with the help of browsers.

For our system, we have chosen to display a browser of all available nodes and links. The user interface of the browser is shown in figure 3.23. The browser is divided into three window-panes, the left-most pane displays all the text nodes available, the right-most pane displays all the picture nodes available, and the middle pane displays the links or references attached with a node. When a user clicks on a node shown in the left-most or right-most pane, the links attached with that node are shown in the middle window-pane.

To construct these window-panes, we have to depend upon the *View* classes of frame-
A list of text nodes

A List of Links

A list of picture nodes

Arrows indicates node "HyperText" is linked with two nodes

Figure 3.23: The Browser interface windows.

works. Class HTBrowserView is derived from the View class of the underlying framework, refer figure 3.24. The constructor of this class takes document, node and link objects as arguments. An instance of this class is created by the document class (discussed in next section).

In our system requirements, we have specified that each node is typically the name of a file, and text nodes are stored in a textnode directory, and picture nodes in a picnode directory. This decision is being taken because all platforms support the ANSI C stan-

![Class Diagram](image)

Figure 3.24: Class diagram for HTBrowserView.
standard for reading and writing files in a directory. The class HTBrowserView has a typical requirement of displaying all the available nodes in textnode and picnode directories. In this class, we will define methods ReadTextDirectory and ReadPicDirectory for reading file names in a directory and displaying them on appropriate windows. The node and link objects will actually be used to show the links available for each node.

### 3.3.3 Application and Document classes

The hypertext program naturally needs its own HTApplication and HTDocument classes. These classes are needed for overall organization of the program, discussed in section 3.2.1. These are specialized versions of Application, and Document classes of the ET++ library (or TApplication and TFileBasedDocument classes of the MacApp library).

MacApp and ET++ will use class HTDocument to save the state of an application. Logically, a document object in these frameworks serves as a heterogeneous repository of anything an application needs to save between executions. For example, a document for the hypertext system might represent a network of nodes arranged in some order, regarding a specific topic.

Similarly, a domain-specific implementation of HTApplication class must be provided. Only a single instance of this class can be created for a program in both ET++ and MacApp. This object may use several, possibly different, document objects. An HTApplication object will be created whenever the hypertext program will be launched or the user selects New/open menu items in the File menu. The HTApplication object will create document objects, explained below.

#### HTApplication class

The program will use only one instance of class HTApplication - a minor specialization of the underlying framework's Application class. This class will need to override the DoMakeManager (or DoMakeDocument) method of ET++. The main methods of the hypertext-application class are:

```cpp
class HTApplication : public Application {
```
The principal work in this class is done by methods `DoMenuCommand` and `DoMakeManager`. Whenever a hypertext program begins, a text editor is displayed by default. In the `File` menu of a text editor, there would be the option of `New` (new text-editor) and `Picture Editor`. Subject to a user demand, the text or picture editor will be opened.
Manager* HTApplication::DoMakeManager(Symbol type) {
    if(type == cHTPicType)
        // document object for picture editor
        return new HTDocument(textpath, picpath, PIC);
    else
        // document object for text editor
        return new HTDocument(textpath, picpath, TEXT);
}

**HTDocument class**

The class **HTDocument** is derived from class **Document** of ET++ (or **TDocument** of MacApp). This specialized class of **Document** builds a display structure and owns an instance of class **HTNode**; this instance should be used to pass various requests to the **HTNode** class. For example, methods **DoRead** and **DoWrite** should pass messages for reading and writing to **HTNode** class.

There is a valid reason for creating view and data-model objects in this class. It is a better idea to keep these objects with the document rather than with the view that displays it, simply because views are transitory, whereas documents exist for the lifetime of the model.

- This class will create different view objects, as per requirements. In the method
DoMakeContent, depending upon the user request, it will create text or picture view objects.

- This class will create an instance of class HTNode for handling a data-model of the hypertext.

- It is responsible for specialized “Hypertext” menus. The menu requirements of a text editor and picture editor are different. This class will provide the method DoMakeMenuBar. This method must handle the set-up of the menus, depending upon the type of node.

- This class will also provide methods to handle file management activities. The methods DoRead and DoWrite must be defined. These methods will ask a node object to save or load its data.

Figure 3.26 shows the responsibilities of the hypertext-document class. The interaction between view and document classes is important for creating displays and getting feedback from users.

### 3.4 Interaction among hypertext objects

So far, we have discussed the data and command-handler classes. In this section, we discuss how hypertext objects interact to provide services to the user. This interaction should explore various collaboration between objects. It is worth discussing such collaboration before we finish design and start implementation because it will give a broader understanding of the classes defined up to now. Also, it may result in identification of other responsibilities and useful classes.

#### 3.4.1 Interaction among application and document objects

The application and document objects are mainly responsible for the overall organization of a program under MacApp and ET++ frameworks. A user creates one instance of class HTApplication and in the implementation of this class, document objects are created. According to system requirements, a user must supply the type of document that is to
DoMakeManager Method

```cpp
HTApplication::DoMakeManager(...type..)
{
    if(type = PIC)
        return new HTDocument(...PIC);
    return new HTDocument(...TEXT);
}
```

DoMakeContent Method

```cpp
HTDocument::DoMakeContent(...type..)
{
    if(type = TEXT)
        new HTextView(.....);
        new HTextNode(.....);
    else
        new HTPicView(.....);
        new HTPicNode(.....);
}
```

Figure 3.27: Object diagram showing interaction between HTDocument and HTApplication classes.

be created, i.e., text or picture. By default, a document object is text, see figure 3.27. A
document object will create view and data model objects.

### 3.4.2 Interaction among document, view and data model classes

The constructor of the HTDocument class creates instances of link cluster classes (HTLink
and HTLinkManager); and these objects will be used by the HTNode and HTView classes.
When creating instances of any particular view or node class, the document class will
supply link cluster objects as parameters.

The application object supplies the document class with the type of node a user
wants to create. Depending upon this type, the HTDocument class method DoMakeContent
either creates instances of HTextView and HTextNode classes or HTPicView and HTPicNode
classes, see figure 3.28.

The document class creates objects of different types. The data model and view classes
interact to provide the services to the user. The user actions on an active window are
passed as commands to the view class. The view classes collaborate with data classes to
provide necessary services. A user interacts with the program through mouse or menu
options.

### 3.4.3 Handling menu and mouse in hypertext system

MacApp and ET++ provide virtual methods in view classes to handle mouse actions and
menu options; refer to section 3.2.2. We will have to override some of these methods in
A document object may create a text or picture view. The handling of mouse or menu actions in text and picture views is different.

**Handling mouse in text editor**

A text editor provides the capability of reading and writing onto its view. Most of the required functionality for a text view is inherited from the framework’s `TextView` (or `TTextView`) class.

```c++
// class HTextView is derived from TextView
// it passes evthandler, window area, and text as parameters
HTextView::HTextView(evthandler* evt, Rectangle* rec, Text* text,
                      HTNode* node,HTlink* link) : TextView(evt,rec,text);
```

The capability of “do-undo-redo” typing is provided by the text view class of the underlying framework. So, the mouse clicking events on a text view are direct reuse from the underlying framework.

**Handling menu in text editor**

The menu of a text editor will be tailored to our system requirements. A hypertext user would like to invoke a picture editor from a text editor and vice-versa. The `File` menu
of text editor should have the option of *new picture editor*. Figure 3.29 shows the menu bar of a text editor. Apart from invoking the text and picture editor, there are other menu-based requests which a text editor must handle:

- The user should be able to mark text - highlight a word.

- The user should be able to link two text nodes or a text and a picture node.

- The reader of a hypertext system should be able to move from a text node to another node either by clicking on a highlighted word or through an index menu showing all available links.

The menu-bar of the text editor has four functionalities. The *File* and *Edit* menu provide standard options (*File/New, File/Open, File/Close, File/Load, File/Save, File/SaveAs, File/Quit, File/Print, Edit/UndoTyping, Edit/RedoTyping, Edit/Cut, Edit/Paste, Edit/Copy, Edit/SelectAll*). But the *File* menu will have a special option of *File/New Picture Editor* - open a new picture editor.

The *View* menu provides two options: *View/Mark* and *View/Link*. The mark option will highlight a word as requested by the user. Basically, it will change the font of the
word from normal to bold. The link option will actually link a word (name of node) with the current node.

The Index menu of the menu-bar will display all the links attached to the current text editor. If there are no links, then no action will be taken on the mouse-clicking index option. If there are links available, then on clicking a link, the node of that link name should be opened.

Method DoSetupMenu must be provided in the class HTextView. The mechanisms for file and edit options are provided by the more standard TextView class. Actual menu making methods will be discussed in the implementation section. The class HTextView must also have its own version of the method DoMenuCommand. This method will be called whenever a user clicks mouse in the menu-bar. Each menu item is given a command number. DoMenuCommand method checks against the command number, and takes appropriate action, see figure 3.30.

Handling mouse in picture editor

The main purpose of a picture editor is to display a bitmap image and allow marking of a certain portion of a picture. Once a portion has been marked, a user should be able to link it with some other text or picture node, see figure 3.31.
In the case of the text editor a “mark” is typically a character word, while in the picture editor it is an editable shape. A user should be able to pick an image from the palette of the picture editor and draw the image on the main view. This is similar to the MacDraw type of drawing editors. The handling of mouse actions in a picture editor is more complex than in a text editor. We need to define a few palette items and palette manager classes to understand the working of “marks” in a picture editor. We will discuss this in the following section while explaining the palette class cluster and its interaction with view classes.

Handling menu in picture editor

The main menu-bar handing mechanism in the picture editor is similar to that of the text editor. There are four menu options, see figure 3.31. The functionality and behaviour of the File and Edit menus are exactly the same as in the text editor. The Index menu will display a list of all the links available, while the View menu has the option of “commit”. The function of this commit option is to link a rectangular area (marked area) with a node (text or picture).

The palette view of the editor will have two two options: “arrow” and “rectangle”.

Figure 3.31: Hypertext picture editor.
The document class is also responsible for managing i/o. The standard methods DoRead and DoWrite are implemented in this class. These methods pass the request for saving and loading to the class HTNode, which will provide hypertext specific i/o implementation.

```c++
bool HTDocument::DoWrite(OStream& data) {
    // fb is a pointer to class HTFileBuffer, initialized in the constructor
    fb->SetFileName(data->FullName());
    fNode->PrintOn(fb);
    return TRUE;
}

bool HTDocument::DoRead(IStream& data) {
    if(type == TEXT || type == PIC) {
        fb->Open(data->FullName(), r);
        fNode->ReadFrom(fb);
        fb->Close();
        return TRUE;
    }
    return FALSE;
}
```

**HTextView**

The HTextView class is derived from the TextView class of ET++. The DoLeftButtonDownCommand method is defined to handle a mouse click in the text editor. Since the user of our hypertext system may be an author or a reader, the domain-specific version of DoLeftButtonDownCommand must provide a mechanism to handle different requests. In the case of the end-user being a reader, the mouse click on a highlighted word will open a node to the reader associated with the word. For an author, a mouse click on any word will open a dialog box to take the input from the author (to link this node with some other node).
Another significant responsibility of this class is to make a menu. The class provides the function `MakeMenu()`, which is called by the document object. The domain-specific version of the `DoSetupMenu` method is also provided. This function is called by framework code after a mouse click in the menu-bar and before the menu handling mechanism (which causes menu item lists to “drop down”) is invoked. Also, an “Index” menu is created. The items of this index menu will list the links attached with any particular node (i.e. a list of links).

```cpp
void HTextView::MakeMenu()
{
    Menu* m = new Menu("View", TRUE);
    m->AppendItems(
        'Mark', cMARK,
        'Link', cLINK,
        0);
    return m;
}

void HTextView::DoSetupMenu(Menu* m)
{
    // enable the mark and link items
    m->EnableItem(cMARK);
    m->InableItem(cLINK);
    // inherit DoSetupMenu of the base view class
    View::DoSetupMenu(m);
}
void HTextView::MakeIndexMenu()
{
    // make an index menu and display link items
    // code is similar to MakeMenu() code
    ...}
```
The functions for handling the menu commands should also be defined in this class. The DoMenuCommand of the framework is overridden here. The method will take necessary steps by identifying which menu in the menu-bar is clicked, e.g.:

```cpp
Command* HTextView::DoMenuCommand(int cmd)
{
    if(cmd == cMARK){
        // show mark dialog box, user has clicked the mark item
        HTMarkDialog::ShowDialog(....);
        // mark the word specified
        fNode->PutMark(...);
    }
    if(cmd == cLINK){
        // show link dialog box,
        HTLinkDialog::ShowDialog(....);
        // confirm that a node with the given name exist, and then link the node
        fNode->AddNode(....);
    }
    if(cmd == cINDEX){
        // the user has chosen one of the items in index menu, open specified node
        fLinkManager->ShowLink(.....);
    }
}
```

**HTBrowserView**

The requirement is that the text editor should be divided into the text-view and browser-view, see figure 4.2. The DoMakeContent method of the document class creates instances of both HTextView and HTBrowserView classes and then combines them into one main view object.

The HTBrowserView class is again derived from the View class of ET++. The DoMakeContent method of this class makes three scrollable window-panes to display a list of nodes (text and picture) and references:

```cpp
VObject* HTBrowserView::DoMakeContent()
{
    // make three scrollable window-panes
    VObject *nscroll, *rscroll, *pscroll;
    ...
    return Expander(...nscroll, rscroll, pscroll,...);
}
```
This class also provides methods ReadPicDirectory and ReadTextDirectory. These methods iterate through a list of nodes (text and picture node objects create their own list - fTextNodeList and fPicNodeList), and each of the text and picture nodes are displayed in windows (iterate through the list of nodes and put them in an appropriate window-pane).

```cpp
void HTBrowserView::ReadTextDirectory()
{
    ...  
    HTObjListIterator* ll;  
    // iterate through the list of nodes in text directory  
    ll = new HTObjListIterator(fNode->fTextNodeList());  
    HTItem* anItem;  
    while(anItem = ll->Next()){
        ...  
        nscroll->Add(anItem);  
        ...  
    }
}

void HTBrowserView::ReadPicDirectory()
{
    ...  
    HTObjListIterator* ll;  
    // iterate through the list of nodes in text directory  
    ll = new HTObjListIterator(fNode->fPicNodeList());  
    HTItem* anItem;  
    while(anItem = ll->Next()){
        ...  
        pscroll->Add(anItem);  
        ...  
    }
}
```

The class provides its own DoLeftButtonDownCommand method. When a user clicks the mouse in either the browser-view's text or picture node window-pane, this method is invoked. The name of the node below the mouse position is then read. The function invokes methods, defined in this class, which open a specific node file and read the links attached with the specified file. The list of links is shown in the middle window-pane of the browser view.
Command* DoLeftButtonDownCommand(Point p, Token, int)
{
    // find the window which contains the point p
    ...
    // open the node file and read the links
    // show the links in the references window
}

HTPicView

The responsibilities of the HTPicView class are similar to the HTextView class. But this class is inherited from the View class of ET++. It owns an object of HTPaletteManager class. The method DoLeftButtonDownCommand of this class basically sends a request to its HTPaletteManager object to handle the mouse click.

Command* HTPicView::DoLeftButtonDownCommand(Point p, Token, int)
{
    // pass the responsibility to palette manager class
    Command* aCommand = fPalManager->CreateActionCommand(p);
    return aCommand;
}

The rest of the methods MakeMenu, DoSetupMenu, and DoMakeIndexMenu are similar to that of the HTextView class.

Menu* HTPicView::MakeMenu()
{
    ...
    return m->AppendItems(‘Commit’, cCOMMIT, 0);
}
void HTPicView::DoSetupMenu(Menu* m)
{
    m->EnableItem(cCOMMIT);
    View::DoSetupMenu(m);
}
HTPaletteManager and HTPalettetmes

According to the design, the HTPaletteManager class has to create instances of the HTPalettetmes class and append them to a list; this is done in the constructor.

HTPaletteManager::HTPaletteManager(HTNode* anode)
{
    // link to the collaborating object
    fNode = anode;
    // Build a list of Palettetmes
    fItemList = new HTObjList;
    Rectangle aRect(0, 0, 32, 32);
    HTArrowItem* anArrowItem = new HTArrowItem(fNode,aRect);
    fItemList->Insert(anArrowItem);
    fCurrent = anArrowItem;
    aRect += Point(0, 32);
    HTRectangleItem* anRectItem = new HTRectangleItem(fNode, aRect);
    fItemList->Insert(anRectItem);
}

The functions HandleMouseSelection and Draw (both invoked by the associated PaletteView object) work by iterating through the list of palette items. The function HandleMouseSelection is used to select the current palette tool:

void HTPaletteManager::HandleMouseSelection(Point& selectpoint)
{
    //mouse has been clicked within the palette view,
    //may need to change the current palette selection
    HTPaletteItem* fPal = NULL;
    HTObjListIterator *ll = new HTObjListIterator(fItemList);
    HTPaletteItem* p;
    while(p = (HTPaletteItem*)ll->Next()){
        if(p->Contains(selectpoint))
            fPal = p;
    }
    // if the current and chosen palette item are same, then no change
    if(fPal == fCurrent) return;
    fCurrent->SetHighlight(FALSE);
    fCurrent = fPal; // make new tool item as current item
    fCurrent->SetHighlight(TRUE);
}
The function `CreateActionCommand` in the class is invoked by the `DoLeftButtonDownCommand` method in class `HTPicView`. The `CreateActionCommand` instead passes the request to the current palette item (selected by a user) to create an appropriate command object.

```cpp
Command* HTPaletteManager::CreateActionCommand(Point& viewpoint)
{
    // mouse has been clicked in the main view, need to create a
    // new "command object".
    if (fCurrent) return fCurrent->CreateCommand(viewpoint);
    else return gNoChanges;
}
```

Each `HTPaletteItem` subclass (`HTRectangleItem` and `HTArrowItem`) defines its own version of the `CreateCommand` method. Depending on the type of palette item object, the appropriate `CreateCommand` method will be invoked dynamically. This method, in the respective implementations, will create new command objects. For example, in the `HTRecItem` class, the `CreateCommand` method creates an `HTRecSketcher` command object:

```cpp
Command* HTRectangleItem::CreateCommand(const Point& p)
{
    HTRecSketcher* aCommand = new HTRecSketcher(fNode, p, fMainView);
    return aCommand;
}
```

**Command Classes**

There are two command object classes in our system, `HTRecSketcher` and `HTRecDragger`, both derived from ET++'s `Command` class, see figure 4.3. These classes provide functions for creating rectangular shapes and dragging these shapes in the main view. The constructor of these classes will require view and node objects and the point where the mouse is clicked.

```cpp
HTRecSketcher::HTRecSketcher(HTNode* pnode, Point p, View* view):
    Command(cRecSketcherCommand,RecSketch,eCmdTypeNormal)
{
    // link to the collaborating object
    fNode = pnode;
}
```
fView = view;
// make a rectangle
aRect = Rectangle(p, Point(0,0));
}

These classes have to provide implementations for TrackFeedback and TrackMouse methods. The TrackFeedback method will draw a rectangle between the points a user has pressed the mouse, that is, a visual feedback of drawing a simple outline. The TrackMouse method provides "house-keeping" actions.

void HTRecSketcher::TrackFeedback(Point anchor, Point last, bool)
{
    Point origin = Min(last, anchor);
    Point extent = Max(last, anchor) - origin;
    aRect = Rectangle(origin, extent);
    ::GrStrokeRect(aRect);
}

The Doit() method creates a rectangular shape (put a mark in the picture editor).

void HTRecSketcher::DoIt()
{
    // create a rectangular shape using HTRecShape class
    aShape = new HTRecShape(fFinalPosition, fView);
    fNode->PutMark(aShape);
}
The undo and redo methods remove the rectangular shape from, or return the rectangular shape to the document:

```cpp
void HTRecSketcher::UndoIt()
{
    fNode->RemoveMark(ashape);
}
void HTRecSketcher::RedoIt()
{
    fNode->PutMark(ashape);
}
```

### 4.2.2 Hypertext data-model classes implementation

**HTNode**

A HTNode is responsible for handling node based operations. It has to create separate lists for text and picture nodes. The constructor of the class will create these list objects.

```cpp
HTNode::HTNode(HTLink* link, char* textpath, char* picpath)
{
    // make a copy of text and picture pathnames
    fTPath = textpath;
    fPPath = picpath;
    fLink = link;
    // create list for text and picture nodes
    fTNodeList = new HTObjList;
    fPNodeList = new HTObjList;
    // update the list by reading directories
    ReadDirO;
}
```

Whenever the program starts, it should provide functions for reading the text and picture directories into the above created lists, which is done by the ReadDir method of the class.

```cpp
void HTNode::ReadDir()
{
    DIR* dirp = opendir(fTPath)
    struct dirent* dp;
```
for(dp = readdir(dirp); dp != NULL; dp = readdir(dirp)){
    ...
    fTNodeList->Insert(someitem);
}
closedir(dirp);
//next same steps for picpath - use fPPath
...

And these lists must be destroyed by the destructor for managing storage.

HTNode::~HTNode()
{
    // destroy the lists
    if(fTNodeList){
        fTNodeList->RemoveAllItem();
        SafeDelete(fTNodeList);
    }
    if(fPNodeList){
        fPNodeList->RemoveAllItem();
        SafeDelete(fPNodeList);
    }
}

The document class actually creates instances of HTextNode and HTPicNode classes.

HTTextNode::HTTextNode(HTLink* li, char* tpath, char* ppath)
    : HTNode(li, tpath, ppath)
{
    // create a list of text marks
    fMark = new HTextMark;
}

HTPicNode::HTPicNode(HTLink* li, char* tpath, char* ppath)
    : HTNode(li, tpath, ppath)
{
    // create a list of picture marks
    fMark = new HTPicMark;
}

The methods ReadFrom and PrintOn are virtual in the class HTNode. The HTextNode and HTPicNode classes provide specific implementations. For example, below is the implementation of text nodes and their interaction with other objects. The node specific
classes first have to read/write the length of the string in the file, followed by the actual text string. Then it will read/write information about links associated with a node.

```cpp
void HTNode::ReadFrom(HTFileBuffer* fb)
{
    // virtual function and uses link object to read links
    fLink->ReadFrom(fb);
}
void HTNode::PrintOn(HTFileBuffer* fb)
{
    // virtual function and uses link object to write links
    fLink->PrintOn(fb);
}
void HTextNode::ReadFrom(HTFileBuffer* fb)
{
    // read data from a file. First thing written in a hypertext file is
    // the length of the file, which maybe read by using: fb->ReadInt()
    char* filestring = new char[fb->ReadInt() + 1];
    fb->ReadData(filestring);
    // read links
    HTNode::ReadFrom(fb);
    //highlight the name of the links in the node
    Text* text = fMark->MakeMark(filestring);
    //set the text into the textview
    fTextView->SetText(text);
    //free filestring
    delete [] filestring;
}
void HTextNode::PrintOn(HTFileBuffer* fb)
{
    // get the data from the text editor
    Text* text = fTextView->GetText();
    //first write the len of the text string
    fb->WriteInt(strlen(text->AsString()));
    //write the text itself
    fb->WriteData(text->AsString());
    //write the links
    HTNode::PrintOn();
}

HTMark, HTextMark and HTPicMark

An HTMark object owns a list of marks. Like the HTNode class, this class is also initialized by its subclasses, i.e., HTextMark and HTPicMark. The base mark class handles operations
for adding and deleting marks from the list, and these requests are passed by node objects.

```
HTMark::HTMark()
{
    // create a list of marks
    fMarkList = new HTObjList;
}
HTextMark::HTextMark() : HTMark()
{
}
HTPicMark::HTPicMark() : HTMark()
{
}
```

The difference in the subclasses is in the implementation of the MakeMark method. This method is dynamically invoked by node and view objects. For example, in ET++, marking of a word is done by putting the "@B" symbol before and after the word which is to be in bold type.

**HTLink and HTLinkManager**

The implementation of HTLink and HTLinkManager classes is quite straightforward. The constructor of the HTLink class constructs a list for the links associated with any node. Actually, link objects are created by the document object, and the document object also creates nodes. Thus, the document class manages both link and node objects. The responsibilities of this class involve reading and writing the links in a specified file. This class eventually passes any such request to the HTObjList class to read and write its links:

```
HTLink::HTLink()
{
    //construct a list of links
    fLinkList = new HTObjList;
}
void HTLink::PrintOn(HTFileBuffer* fb)
{
    //bureaucracy...list write yourself
    fLinkList->PrintOn(fb);
}
void HTLink::ReadFrom(HTFileBuffer* fb)
```
The HTLinkManager class constructor is given the pathnames of directories which contain text and picture node files. We decided to implement this class because the responsibility of this class is very unique. It has to deal with application and document objects to open a new document. It has a method ShowLink, which is invoked by text and picture view objects (when the user moves from one node to another via a link).

```cpp
HTLinkManager::HTLinkManager(char* textpath, char* picpath)
{
    //make a copy of path names
    fTPath = textpath;
    fPPath = picpath;
}

void HTLinkManager::ShowLink(Application* myapp,
                              HTDocument* doc, char* link)
{
    //if the document is already open then show it
    if(doc)
        doc->Show(); // method defined in framework document class
    else{
        //depending upon the type of doc(text or pic),
        //form a complete filename
        char* nshow;
        if(doc->Type() == TEXT)
            nshow = form("%s/%s",fTPath,link);
        else
            nshow = form("%s/%s",fPPath,link);
        myapp->OpenDocument(nshow); //method defined in application class
    }
}
```

4.2.3 I/O, Container, and Dialog classes

HTObjList and HTObjListIterator

Class HTObjList and its associate Link and HTObjListIterator classes implement a generic linked list. The important aspect of this class is saving and loading links into a
file. The methods ReadFrom and PrintOn are defined in this class and invoked by the HTLink object. We have implemented PrintOn in a way that it first writes the number of articles in the list into the file, followed by the length and name of each item. The ReadFrom method is complementary to PrintOn.

```cpp
void HTObjList::PrintOn(const HTFileBuffer* fb)
{
    //write the length of the list into file buffer
    fb->WriteInt(Length()); //length method of class returns length of list
    
    //iterate through the list and write length and name of each list item
    HTObjListIterator iter(this);
    char* anItem;
    while(anItem = iter.Next()){
        fb->WriteInt(::strlen(anItem));
        fb->WriteChar(anItem);
    }
}
void HTObjList::ReadFrom(const HTFileBuffer* fb)
{
    // read the length of list to be created
    short length = fb->ReadInt();
    
    // clear the old list
    RemoveAllItem();
    // length shouldn't be zero to create a new list
    if(length!=0){
        for(short i=0; i < length; i++){
            // read the length and item from the file and insert it in the list
            
            Insert(someItem);
        }
    }
}
```

HTFileBuffer

The class HTFileBuffer implements simple functions, such as open and close file operations. The class is initialized by the document object, and later an object of this class is passed to HTLink class.

```cpp
FILE* HTFileBuffer::Open(char* file, open_mode om)
```
// switch statement according to open_mode type
switch(om){
case r:
    Data = fopen(file, '"r"');
    break;
case w:
    Data = fopen(file, '"w"');
...
...}
opened = TRUE;
return Data;
}

void HTFileBuffer::Close()
{
    if(opened){
        if( fc = (fclose(Data)) == 0)
            opened = FALSE;
        else
            // print warning message
    }
}

The other functions in this class are simple to implement. For example, a ReadData function is supposed to read the file string:

int HTFileBuffer::ReadData(char* fd, short l)
{
    if(opened){
        int ec = fread(fd, sizeof(char), l, Data);
        if (ec < 0)
            return -1;
        else
            return ec;
    }
    else
        return EOF;
}
HTMarkDialog and HTLinkDialog

The dialog classes are derived from ET++’s Dialog class. The make function in these classes is DoMakeContent, which constructs the dialog boxes.

VObject* HTMarkDialog::DoMakeContent()
{
    // make two action buttons 'ok' and 'cancel'
    VObject *fMarkDialog = new HBox(10,
        new ActionButton(cIdOk, 'Ok', TRUE),
        new ActionButton(cIdCancel, 'Cancel', TRUE),
        0);
    //construct a dialog box with textfield and combine it with fMarkDialog
    return new Matte(
        new VBox(10,
            new TextItem("Mark:") , markField = new TextField(cIdNone)
        fMarkDialog,
        0),
    );
}

Class HTLinkDialog also defines its own implementation of DoMakeContent. Both the classes have a method ShowDialog defined in their respective implementations. This method is invoked by view objects, when a user clicks the menu item ‘mark” or “link”. The role of this function is to show the dialog box on the screen.

int HTLinkDialog::ShowDialog(Clipper* cp)
{
    //calls ShowDialog method defined in ET++ dialog class, which will
    //show the dialog box on the screen.
    return ShowDialog(cp);
}

4.2.4 ET++ hypertext example

Figure 4.4 shows the implementation of the hypertext text editor program, while 4.5 shows that of the picture editor program.

We have constructed a small hypertext system on the geography of “India” to test our system. There are about 30 text and picture nodes. As shown in figure 4.4, the menu-bar has four options. The “Index” menu contains a list of nodes which are connected with
Country - India - Geography

Long-form Name: Republic of India
Type Federal Republic
Capital: Delhi
Area: 3, 287, 590 km-square
Land Area: 2, 973, 190 km-square
Comparative Area: Slightly more than one-third the size of the US.

Land Boundaries: 14,103 Km total: Bangladesh 4,053 km, Bhutan 605 km, Burma 1,463 km, China 3,380 km, Nepal 1,690 km, Pakistan 2,912 km.

Coastline: 7,000 km
Contiguous Zone: 24 nm;
Climate: varies from tropical monsoon in south to temperate in north.

Economy: India's Malthusian economy is a mixture of traditional and village farming and handicrafts, modern agriculture, old and new branches of industry, and a multitude of support services.

People:
Population: 849,746,001 (July 1990), growth

Figure 4.4: Implementation of hypertext text editor and browser.
Figure 4.5: Implementation of hypertext picture editor.
this node. Also, the links are highlighted (marked nodes). The user can open any node either by choosing an item from the index menu or by clicking any highlighted word. The “View” menu has options for showing a mark or link dialog box.

Similarly, the figure 4.5 shows a picture image of the map of India. The palette has two tools; arrow and rectangle. The rectangle item can be used to mark a region in the picture image and the arrow item can be used to drag the rectangle anywhere on the main view.

The browser view in the text editor has three window-panes. The left-most pane shows a list of text nodes, while the right-most pane shows a list of picture nodes. A mouse click on any one of these windows will result in the display of links attached to the corresponding node under the mouse position in the middle window-pane.

The user interface of our hypertext program is simple. All the functionalities of the system are self explanatory. Both the views are expandable. The text editor provides a vertical scroller for vertical scrolling, while the picture editor provides both horizontal and vertical scrollers. The dialog boxes for open/load mechanism are the standard dialog boxes of ET++.

### 4.3 MacApp based Hypertext implementation

During the design stage of hypertext classes in chapter 3, we identified the commonalities in ET++ and MacApp frameworks. Consequently, we designed the hypertext classes so that they may be ported onto both frameworks. But naturally, there will be changes at the implementation level. In this section, we discuss the MacApp based implementation of hypertext program.

#### 4.3.1 C++ construct for MacApp

Before looking at the hypertext implementation, we shall look into the MacApp implementation. There are a few extensions to C++ for use with MacApp, and quite a number of anomalies in the C++ MacApp classes. A full discussion of extensions and anomalies in MacApp is beyond the scope of this research, albeit we will discuss some of them, relevant to this work.
MacApp started off as an Object Pascal class library that was closely integrated with the underlying Macintosh Toolbox and Operating System. All data members in MacApp are public (no mechanism for encapsulation) and methods of classes are implicitly virtual (all methods in classes may be overridden).

There are no constructors; new instances of classes are initialized by calls to explicit initialization methods, such as Initialize(). This method is a kind of substitute constructor. Similarly Free() method is used as the destructor under this framework.

MacApp, like ET++, does not use multiple inheritance.

Those member functions that are defined by MacApp must follow Object Pascal conventions rather than C++ conventions. These functions must be qualified by the keywords "virtual pascal", e.g.:

```pascal
class TMyView : public TView {
public:
    ...
    virtual pascal void DoMouseCommand(VPoint& themouse, TToolboxEvent *event, CPoint hysteresis);
    ...
    virtual pascal void Draw(const VRect& somearea);

    ...
private:
    // private attributes ....
}
```

Object pascal's inherited keyword has been added to this C++ dialect. The point of using this keyword is that the function defined with inherited is derived from its parent class, where the function is defined.

```pascal
pascal void UTEDocument::Free()
{
    ...
    inherited::Free();
}
```
4.3.2 Implementation of Application and Document classes

Naturally, the MacApp based hypertext system will have its own Application and Document classes. These are minor specializations of the TApplication, and TFileBasedDocument classes as provided in MacApp library.

The Application object is created by the main function of the program.

```cpp
void main()
{
    THTApplication* aHTApplication;
    InitToolBox(); // MacApp tool box
    if (ValidateConfiguration(gConfiguration)){
        InitUMacApp(8);
        InitUPrinting();
        // other initializations
        ...
        ...

        aHTApplication = new THTApplication;
        aHTApplication->IHTApplication();
        aHTApplication->Run();
    } else
        StdAlert(phUnsupportedConfiguration);
}
```

A THTApplication has an initialization method, THTApplication::IHTApplication(), which establishes a proper MacApp "Finder interface". Its DoMakeContent() method
creates instances of class \texttt{THTDocument}. The design of class \texttt{THTApplication} is shown in figure 4.6 and the main code is:

\begin{verbatim}
pascal void THTApplication::IHTApplication()
{
    this->IApplication(kFileType, kSignature);
}
pascal TDocument* THTApplication::DoMakeDocument(CommandNumber aCommand,
            TFile* itsfile)
{
    THTDocument* aDocument = new THTDocument;
    if(aCommand == cHTPicType)
        aDocument->IHTDocument(itsfile,kSignature,PIC,textpath,picpath);
    else
        aDocument->IHTDocument(itsfile,kSignature,TEXT,textpath,picpath);
    return aDocument;
}
\end{verbatim}

The responsibilities of a document object are shown in the figure 4.6. Some examples of the member functions of class \texttt{THTDocument} are:

\begin{verbatim}
pascal void THTDocument::IHTDocument(TFile* itsFile, OSType itsCreator,
            doc_type* type,char* textpath,char* picpath);
{
    this->IFileBasedDocument(itsFile, itsCreator);
    fType = type; // type of document requested
    fTPath = textpath;
    fPPath = picpath;

    THTLink* aLink = new THTLink;
    aLink->IHTLink();
    fLink = aLink;

    THTLinkManager* aLinkManager = new THTLinkManager;
    aLinkManager->IHTLinkManager(fTPath, fPPath);
    fLinkManager = aLinkManager;

    MakeNode();
}
\end{verbatim}

The \texttt{MakeNode} method of this class has to make text and picture node objects:
pascal void THTDocument::MakeNode() {
    ... 
    if(type == Text){
        fNode = new HTextNode(...);
        fNode->IHTTextNode();
    } else{
        fNode = new HTPicNode(...);
        fNode->IHTPicNode();
    }
}

Similar to the ET++ implementation, a MacApp document object will also create view objects. The method DoMakeViews is a replica of the DoMakeContent of ET++ document object.

pascal void THTDocument::DoMakeViews(Boolean)
{
    TWindow* aWindow;
    ... 
    if(fType == Text){
        // create textview and browser view objects
        ... 
        THTextView* aTView = (THTextView*)aWindow->FindSubView('HTEX');
        fTView = aTView;
        fTView->IHTTextView(fNode, fLink, fLinkManager);
        ... 
    } else{
        // create picture view and palette view objects
        THTPicView* aPView = (THTPicView*)aWindow->FindSubView('HPIC');
        fPView = aPView;
        fPView->IHTPicView(fNode, fLink, fLinkManager);
        ... 
    }
    ... 
}
4.3.3 Implementation of View and Data-Model classes

View Classes

The implementation of View classes, using MacApp, for the hypertext system is similar to that of ET++. These classes have private data fields and override virtual methods. For example, the THTPicView and THTPaletteView classes detailed design is shown in the figure 4.7. The method DoPostCreate extends the code to establish links to THTNode, THTLink, and THTLinkManager objects.

```pascal
pascal void THTPicView::DoPostCreate(THTDocument* itsDocument)
{
    inherited::DoPostCreate(itsDocument);
    THTDocument* aHTDoc;
    aHTDoc = (THTDocument*) itsDocument;
    fPalManager = aHTDoc->PaletteLink();
    fNode = aHTDoc->NodeLink();
}
pascal void THTPicView::Draw(const VRect& area)
{
    //node draw yourself
    fNode->Draw(area);
}
pascal void THTPicView::DoMouseCommand(VPoint& themouse, TToolboxEvent*, CPoint)
{

```
TCommand* aCommand;
aCommand = fPalManager->CreateActionCommand(theMouse);
if (aCommand) PostCommand(aCommand);
}

As in the ET++ implementation of the HTextView class, the THTextView class is derived from the TTEView class of MacApp. The data owned and responsibilities of this class are:

class THTextView : public TTEView{
public:
    void ITextView(THTNode* aNode, THTLink* aLink,
                   THTLinkManager* aLinkManager);
    pascal void DoMenuCommand(CommandNumber aCommandNumber);
    pascal void DoSetupMenus();

private:
    THTNode* fNode;
    THTLinkManager* fLinkManager;
    THTLink* fLink;
};

Method ITextView copies the objects passed to it into the private fields of the class. These objects are used by the DoMenuCommand method to pass messages.

void THTextView::ITextView(THTNode* aNode, THTLink* aLink,
                            THTLinkManager* aLinkManager);
{
    fNode = aNode;
    fLink = aLink;
    fLinkManager = aLinkManager;
}

pascal void THTextView::DoMenuCommand(CommandNumber aCommandNumber)
{
    ...
    switch(aCommandNumber){
    case MARK:
    ...
        fNode->PutMark(...);
    break;
    case Link:

115
Node, Link, and Mark classes

There are no major changes in the MacApp implementation of the THTNode, THTLink, and THTMark classes from the ET++ implementation. The design of these classes can be referred from section 3.3.1. The responsibilities and data owned by these classes is the same as in ET++. Thus, these are "plug-able" components with minimal or no changes required.

One extra method that has been defined in these class clusters is that all these classes have to define their own constructor initialization methods, e.g.:

```cpp
void THTNode::IHTNode(THTLink* alink, char* textpath, char* picpath)
{
    //make copies of text and picture pathnames
    fTPath = textpath;
    fPPath = picpath;
    fLink = alink;
    //create list for text and picture nodes
    HTObjList* aTNodeList = new HTObjList;
    aTNodeList->IHTObjList();
    fTNodeList = aTNodeList;
    HTObjList* aPNodeList = new HTObjList;
    aPNodeList->IHTObjList();
    fPNodeList = aPNodeList;
    //update the list by reading directories
    ReadDir();
}
```

Command Objects

The implementation of Command objects is similar in MacApp to ET++. According to our design requirements, we need THTRecSketcher and THTRecDragger command classes
to draw and move a rectangular shape in the picture editor. These classes are derived from the TTracker class of MacApp. The detailed class hierarchy is shown in figure 4.8.

As in the ET++ implementation, the initialization functions of the these classes will require links to collaborating document, view, and node objects, e.g.:

```cpp
void THTRecSketcher::IHTRecSketcher(TDocument* doc, THTNode* anode,
                                    THTPicView* view, const VPoint& startpoint)
{
    ITracker(cNodeSketcherCommand,
             doc, kCanUndo, kCausesChange,
             doc, view
             view->GetScroller(TRUE);
             startpoint);

    //make links with node and view objects
    fNode = anode;
    fView = view;
    //make a rectangle
    aRect = CRect(startpoint, CPoint(0,0));
}
```

Similarly, we need to implement the methods TrackMouse and TrackFeedback. The
objective of these methods is the same as in ET++. The TrackMouse method is called by the application object, and is basically intended as house keeping activity required by THTPicView object, while TrackFeedback is intended to provide on-screen feedback for the user while the mouse is being tracked.

```pascal
void THTRecSketcher::TrackFeedback(TrackPhase, const VPoint&, const VPoint&, const VPoint& nextPoint, Boolean mouseDidMove, Boolean)
{
    if(mouseDidMove){
        CPoint acenter = fView->ViewToQDpt(nextPoint);
        ::Pt2Rect(acenter, acenter, aRect)
        ::InsetRect(aRect, -12, -12);
        ::FrameOval(aRect);
    }
}
```

The rest of the operations in command classes are implemented identically to the ET++ implementation. For example, the DoIt, RedoIt, and UndoIt methods are implemented identically.

```pascal
void THTRecSketcher::DoIt()
{
    aShape = new THTRecShape;
    aShape->IShape(fFinalPosition, fView);
    fNode->PutMark(aShape);
}
void THTRecSketcher::UndoIt()
{
    fNode->RemoveMark(aShape);
}
void THTRecSketcher::RedoIt()
{
    fNode->PuMark(aShape);
}
```

I/O and Container classes

The HTObjList and HTFileBuffer class clusters, which were explained in section 3.5.3, implemented in ET++ and MacApp without any major change.
The \texttt{THTDocument} object defines \texttt{DoRead/DoWrite} methods to support reading and writing the state of the hypertext application. There is no change in the algorithms for this between the ET++ and MacApp implementations, see section 4.2.3.

\section*{Dialog Implementation}

The implementation of \texttt{THTLinkDialog} and \texttt{THTMarkDialog} classes in MacApp is based on the same design as used within ET++. Here, we derive our two classes from the MacApp based dialog class - \texttt{TDialogView}, see figure 4.9. The Macintosh applications can use one of two kinds of dialogs: modeless and modal. A modeless dialog is presented in a window that can be used and then placed behind another window. A modal dialog on the other hand is one to which the user must respond in some way and explicitly dispose of before she can continue running the application [Wilson et al. 90].

We need the modal view, as we need feedback from the user about the word or area she wants to mark, or for creation of a link between two nodes. So, we have to override the \texttt{ShowModalDialog} method of \texttt{TDialogView} class.

\begin{verbatim}
pascal int THTMarkDialog::ShowModalDialog()
{
    TWindow* aWindow;
}
\end{verbatim}
IDType  dismisser;
CStr255  selection;
TEditText*  aEditText;
TStaticText*  aStaticText;

//get the selection from user
fView->GetSelectionString(selection);
FailNil(aWindow = gViewServer->NewTemplateWindow(1003, NULL);
...
  dismisser = aWindow->PoseModally();
...
  aWindow->CloseAndFree();
  if(dismisser == 'ok ')
    return TRUE;
  else
    return FALSE;
}

### 4.3.4 MacApp Hypertext example

Figure 4.10 shows the implementation of hypertext based text editor program with MacApp, while figure 4.11 shows that of the picture editor program.

As in the ET++ hypertext example, here also we have constructed a simple geography of “India” to test the hypertext system. The menu-bar in both the editors have the same four options of File, Edit, View, and Index. The “Index” menu item shows a list of links to which a particular node is linked. Also, all the “mark” or highlighted words indicate the links attached with a node. The links may be activated either by clicking these marked words or through the list in the Index menu.

The browser shows the list of nodes (both text and picture) and references attached to each node. The dialog-box may be shown by using the View menu, which will display different dialog boxes for marking and linking.

Thus we have implemented the same hypertext system using both the MacApp and ET++ frameworks. Now, we will evaluate the design of our class clusters from the reusability and changing requirements points of view.

120
Figure 4.10: Screen dump from MacApp example of hypertext based text editor.
Figure 4.11: Screen dump from MacApp example of hypertext based picture editor.
4.4 Evaluation

To validate that our design is good, we must evaluate the system. A good design is easy to
maintain, easy to change, and confirm to the requirements of the system. A well designed
system will endure these properties gracefully; a poorly-designed system will fall against
these characteristics [Coplien92].

Our initial aim was to construct a cluster of classes which model a hypertext program
and can be plugged into two different application frameworks, running on different plat­
forms. During the design phase, we focused on the commonalities in MacApp and ET++
particular to hypertext requirements. The success of designing such a “plug-able” class
clusters depends upon the following factors:

- Design validation for reuse,
- Adding new functionality, and
- Changing requirements.

In this section we will discuss all these issues, and will identify the strength and
weaknesses of our plug-able hypertext class clusters.

4.4.1 Design Validation for Reuse

MacApp and ET++ are based on the object-oriented paradigm, and the main aim of
these application frameworks is to provide reusability, basically with the help of class
inheritance concepts. The idea of building class clusters, which embodies some domain­
specific knowledge, to port on top of these application frameworks can enhance reusability
considerably. Keeping this in mind, we have designed and implemented the hypertext
cluster of classes. The potential degree of reuse may be directly determined by: the
number of frameworks on which the class clusters can be ported, and how difficult it is
to adopt these clusters with different frameworks. Since we are limited to MacApp and
ET++, we will consider the reuse from these two frameworks view points.
The Data-model classes

Booch [Booch91] suggests that the two most meaningful metrics to measure a good design are: coupling and cohesion. Though these are borrowed notions from structured design, but with a liberal interpretation, they also apply to object-oriented paradigm. We will validate our the data-model design from these notions.

The data model classes are designed and implemented in such a way that we can pick them “off-the-shelf” and port them with MacApp and ET++. Of course, where necessary we have to change the implementation style. The functionality and data owned by HTNode, HTLink, HTMark, HTPalette, HTFileBuffer, and HTObjList class clusters remained the same for both implementations. In the MacApp implementation, many of the classes have to define extra methods to support class initialization.

There is strong interaction between these class clusters to provide domain-specific services, that is, the cohesion between objects of these classes is strong. Since they model a hypertext program, such cohesion is perfectly valid. For example, HTNode, HTLink, and HTMark class clusters pass messages to each other to support hypertext requirements. Similarly, HTFileBuffer and HTObjList are i/o and container component classes respectively, and model hypertext specific i/o.
There is very weak interconnection between these data-model classes and the underlying framework, which indicates that these classes support reusability and may serve different clients, see figure 4.12. Thus, we are able to de-couple our data-model classes from MacApp and ET++.

The View classes

The specialized View classes in our system are designed to represent different nodes in some graphical form. We have designed and implemented HTextView and HTPicView classes to display text and picture editors respectively. The HTPicView (or THTPicView) class is derived from the View class of the underlying framework, while HTextView class is derived from the TextView class (the THTextView class is derived from the TTEView class in MacApp).

Although the HTPicView class re-uses the common functionalities available in the View classes of both frameworks, the HTextView class has some problems which hamper reusability. Since hypertext specific text-view classes are derived from the TextView class (in ET++) or TTEView class (in MacApp), some of the functionality and the way these classes behave are framework dependent. The problems and their solutions are listed below:

• The Problems with HTextView class:

  1. The method of highlighting a word in ET++ and MacApp is different because TextView (or TTEView) depends not only on the View class of underlying framework, but also on the Text class of framework.

    // the ET++ TextView highlights a text using following lines

    Text* atext = new StyledText(gFixedFont, "*characterstring");
    SetText(atext);

    // the MacApp TTEView highlights a text using following lines
The program specific textview classes are inherited from underlying framework specific textview class. The reuse is hampered.

In new design, the program specific textview is from view class of underlying framework. It also owns a data field to framework specific textview class. The reuse is improved as the interface of methods in both implementation will remain same.

Figure 4.13: Class diagram representing dependency of TextView classes.

```
TTEStyleCommand* aCommand = DoMakeStyleCommand(newStyle, 
cStyleChange, doFace + doToggle);

fHTDoc->PostCommand(aCommand);
```

2. The interface of a few methods change between ET++ and MacApp because the attributes of functions change due to dissimilarities between the TTEView and TextView classes. This in fact changes the implementation of a few methods in the HTNode class.

- **The Solution:** A possible solution to this problem might be to change the class hierarchy of the HTextView class. Instead of direct specialization of HTextView class from the TTEView or TextView classes, we can inherit it from View class of the underlying framework, see figure 4.13. To use the methods available in the TTEView and TextView classes, we can create an instance of these classes. This object may
be used later to inherit the specialized behaviour of the underlying framework's `TextView` class. This will improve reuse of hypertext classes and software will be easy to port across the platforms.

The above development reflects the fact that the design of an application is not completed by implementing the system. Rather, we may have to redesign our system after implementing it to provide adequate reusability. Thus, it supports the idea of the iterative-design approach in an object-oriented environment.

The `HTBrowserView` and `HTPaletteView` classes are again derived from the `View` class of the underlying framework and they are ported onto MacApp and ET++ with slight modifications in implementation.

The Document and Application classes

Both MacApp and ET++ based hypertext programs define their own versions of `Application` and `Document` classes. For all framework based programs, these are usually the first two classes which are designed and implemented, as they organize the overall structure of a program. There is no hard-and-fast rule for this, though. The `Application` and `Document` classes are at a higher level of abstraction under both frameworks, and the duties of classes derived from these classes are similar. That is, the roles of the `THTApplication` and `HTApplication` classes are identical. Similarly, the roles of the `THTDocument` and `HTDocument` classes are identical.

In both MacApp and ET++, we have identical methods in the `HTApplication` class, the only difference is that in the MacApp `THTApplication` class we have extra lines for other classes and MacApp “ToolBox” initialization calls. The `HTDocument` class defines the `DoMakeContent` method to create various view objects, while `THTDocument` does the same work by overriding `DoMakeViews`. Apart from these changes, there is practically no change in the design of these classes. Thus, we may say that our hypertext document and application classes represent suitable abstraction for design reuse.

4.4.2 Adding New Functionality

One of the most important factors which contribute to the evaluation of a design is its resilience to change. In this section, we will consider a few improvements to the
functionality of the hypertext program and see how it responds to such changes.

- **What if we want to add a new type of node to existing system?**

  The first improvement that comes immediately to mind is how the system design will change if there is a demand for more than a text and picture node based hypertext program. For example, if we want to add a new type of node say, *sound* or *video* node, then what are the changes that have to be brought in the design? This type of additional functionality is not unusual.

  We have to add a new class `HTSoundNode` into our system. This class, like the `HTTextNode` and `HTPicNode` classes, will be a specialization of the `HTNode` class, see figure 4.14. The role of the `HTNode` class is basically to support the saving/deleting of links associated with a node, drawing a node, and editing a node. These functionalities are abstract and are needed by any type of node; class `HTSoundNode` may re-define or reuse them according to need. But things like marking a portion of a sound-track; support for connection between an external device (say, a tape-recorder) and the user interface for sound control panel will be complex. This indicates that we have to derive one more class `HTSoundView` from the `View` class of the underlying framework, which will provide a visual display of a sound control panel on the screen.

  Similarly, we have to specify the directory path-name where information about sound nodes will be stored. This directory name has to be supplied to the application class, and later must be passed to the document class. Also, when the `MakeNode`
method of the document class makes three instances of class HTNode, instead of two (one extra for sound nodes).

Thus, we have to define a couple of classes to support new node types and a few changes in the document and application classes. The rest of the design will remain same.

- **What if we want to add a new tool in the palette box?**

The HTPalette class cluster has two types of palette items, HTArrowItem and HTRecItem, both derived from the HTPaletteItem class. The HTPaletteManager creates instances of these classes and manages the control over palette items (passed to it by HTPaletteView class). The “arrow-item” is used to drag an image on the picture editor, while “rectangular-item” creates command objects for displaying rectangles onto the picture editor.

Suppose we want to add a “circle-item” to create a circular command object; what changes should be brought into the design? It is evident that we will need to define an HTCircleItem class, which would be similar to the HTRecItem class. Most of the design and code we can reuse from HTRecItem class, and there will be minimal effort to bring up a new tool in the palette box.

Similarly, to create a circular image, like a rectangular image, we have to define an HTCircleSketcher class. This class again will be derived from the Command class of the underlying framework. Here also, we will reuse design and implementation of the HTRecSketcher class to build the HTCircleSketcher class.

- **What if we want a “font-menu” in the menu-bar?**

In our current system, we are using the **bold** technique to identify that a particular word in a node is marked, and clicking the mouse on it will activate a link. But there is every possibility that in future we need to add functionality for changing the font to: *italic, underline, outline*, or *shadow*.

To add such functionality, we can depend upon the underlying framework. Both ET++ and MacApp have predefined **Font** classes. We can reuse them and add a font-menu into our currently existing menu-bar. Since the HTextView (or THTTextView)
class is derived from the `TextView` class of the underlying framework, we need not worry about how the fonts will change. It will become the responsibility of frameworks to change the fonts.

Thus, we have added three more functionalities into our hypertext program. To bring these changes, we have defined a few more classes. At the same time we have modified a few functions in the existing design. In all, we have found that changes to initial design are compatible. In other words, we could extend our existing hypertext system in simple ways to impart additional behaviour. Thus, our design is supporting reusability without any fundamental change in the semantics of the abstract class clusters.

### 4.4.3 Changing Requirements

The design of a system should be such that it could sustain the changing requirements. It is quite possible that with a change in requirements, the role of a few classes may change and they may have to be re-defined. But the effect on other classes, which are not involved with this change, should be minimal. Otherwise reuse will surely be hampered. Here, we will study the effect of changing requirements on our design.

- **User wants to display browser with picture nodes**

  It may be argued that why have a node browser only with text based editors? Some users would like to have a browser with both text and picture editors. This additional requirement may be fulfilled by a simple implementation change in the `DoMakeContent` (or `DoMakeViews`) method of the `HTDocument` class.

  Like a text based editor, a picture based editor can also be divided into a picture and browser window. Which basically means that we have to re-define the coordinates of the picture editor window, e.g.:

  ```
  // currently a picture window is ET++ is created as:

  VObject* aWindow = new Expander(cIdNone, eVert,
      Point(2,2), fPalView, fMainView, 0);

  // the aWindow object is made of palette-view and main view objects
  ```
// to add browser-view in this window, we need to add its object

VObject* aWindow = new Expander(cIdNone, eVert,
    Point(2,2), fPalView, fMainView, fBroView, 0);

In addition to this, users may like to switch between “show” and “hide” browser options (i.e., she should be able to see a browser in both editors, with the capability of hiding it). To provide this, we will have to add a few lines of code in the MakeMenu and DoSetupMenu methods of the HTPicView and HTextView classes. Basically, append one more item in the View menu: Hide Browser and enable this option at start of the program.

When a user chooses the “Hide-Browser” menu item from View menu, switch the code to:

VObject* aWindow = new Expander(cIdNone, eVert,
    Point(2,2), fPalView, fMainView, 0);

That is, when the document object receives a command object from the user to hide a browser window, then old window object should be deleted and a new window object should be created without the HTBrowserView object. Thus, the coordinates and objects enclosed in a window should be redefined.

- **User wants to add search mechanism in the text editor**

Since we have an article based system\(^1\) rather than a card-based system\(^2\), the text in a text editor may be very long. To search for a particular word may be painful. So, users may request a searching mechanism.

To provide such a functionality, we first have to provide a menu item for this in the View menu of the main menu-bar. We can add a Find option in the View menu. To do this, we have to modify the MakeMenu and DoSetupMenu methods of the HTPicView and HTextView classes.

---

\(^1\)System that offers scrollable screen.
\(^2\)System that offers only fixed screens
We have to define an HTSearch class. The responsibility of this class will be to search for a word from the supplied text string. And the class will own a pointer to the string which is to be searched.

class HTSearch
{
public:

    HTSearch(char* aString);
    ~HTSearch();
    boolean FindWord(char* aWord);
...
private:
    char* fString;
...
}

The constructor of the HTSearch class makes a copy of the string to be searched. The destructor destroys such a string to free the memory space allocated. The FindWord method can employ some standard string search algorithm.

The DoMenuCommand method in the HTextView class will create an instance of this class and will invoke the FindWord method of the class. Thus, the DoMenuCommand method has to be modified.

Command* HTextView::DoMenuCommand(int cmd) {

    if(cmd == cSEARCH) {
        char* tmp = ((char*)this->GetText());
        HTSearch* aSearch = new HTSearch(tmp);
        ...
        int ec = aSearch->FindWord("someword");
        if(ec == 0)
            // do something
        else
            // do something else
    }
    ...
}
We will need to define one more Dialog class to take the input from the user about the word she wants to search. So, class HTSearchDialog is evident. This class can be designed and implemented using the HTMarkDialog or HTLinkDialog classes. There will be practically no change in the implementation of the HTSearchDialog class, as the behaviour is same as that of other already defined domain-specific dialog classes.

These additional classes are again versatile and can be implemented on top of both ET++ and MacApp. To fulfill the search requirement, we have not changed any class design of the original system, but have added one more compatible component class. The flexibility of adding class clusters with changing requirements supports reuse. Where required, we may have to change implementation methods in the class, but the basic fabric of our design remains intact.

4.5 Further Research

The requirement-specific reusable class clusters we have constructed in this research are based on ET++ and MacApp libraries. There are possibilities of further research in the following directions:

- Like MacApp and ET++, Borland’s OWL (Object Window Library) library is also based on the Model-View-Controller (MVC) philosophy. The same is the case with MicroSoft’s MFC (MicroSoft Foundation Classes) library. Although OWL and MFC libraries are not implemented in exactly same fashion as MacApp and ET++, it would be interesting to build such usable classes which model some specific program on top of OWL and MFC.

- The research supports the iterative-design philosophy in the object-oriented environment. While implementing our program specific THTextView class, we found reuse has been hampered due to over dependence on the underlying framework. We redesigned a few classes to provide higher reusability. This recognizes that design does not stop at coding. The area of providing reuse through the evolutionary approach in object-oriented technology is another field which should be ventured.
There is need to develop tools that assist in measuring the reuse. Tools which can exploit the reuse at all stages of the class hierarchy and software development. The class clusters which model programs, like our hypertext system, have no concrete device to measure the degree of reuse (both design and code). This is another area which should be exploited.

4.6 Summary

In this chapter we have implemented our hypertext design, from chapter 3, onto the ET++ and MacApp frameworks. Where necessary, small changes in the design were made for efficiency and ease of implementation.

While porting the code in MacApp, we were restricted by special constructs of C++, as used in MacApp. There were changes in the implementation of the hypertext program on MacApp. Overall, we found that the generalization of hypertext classes in the previous chapter have reduced the problem of cross-platform porting considerably.

We have divided our cluster of classes into three sections: data-model classes, view-classes and application classes. The data-model classes were found to be generic enough and the degree of reuse in these classes is sufficiently high. The view classes are framework dependent. We isolated common behaviour found in the view classes of MacApp and ET++, and based on these commonalities, we constructed domain-specific reusable view class components. While designing the text-view classes, we found that specializing hypertext specific text-view classes from framework specific text-view classes hampered the reuse. This is because the way low level text-view classes handle the text-based commands differs in the two frameworks. So there are differences in the implementation of hypertext specific text-view classes in ET++ and MacApp. But at the same time, we are reusing text-view functionalities available in each framework, instead of building own text-view class from scratch. The design of document and application classes is same for two frameworks and there are only small changes in the implementation.

We have done a critical evaluation of our design by noting the design modification necessitated by the addition of new functionality. We found the design to be quite stable from the reuse point of view.
Chapter 5

Conclusions

In this thesis we have developed a cluster of classes which represent a hypertext system. This class cluster was designed to provide substantial reusability. We have used the plug-adaptor approach to design the system. Using this approach we were able to learn from our experience and provide some insight into object-oriented development and reuse. In brief, we have presented:

- The problems in the software industry and the role of object-oriented technology to enhance reuse.
- The frameworks ET++ and MacApp are designed to provide reuse through subclassing the available classes in these libraries. The domain specific knowledge can be introduced in application-specific classes and plugged into these frameworks.
- The application-domain classes can be implemented so that they can be ported effectively with different frameworks.
- In designing the hypertext specific classes, we have identified some important processes within the design process; some of them are:
  - Identify commonalities in the ET++ and MacApp frameworks which are specific to the hypertext system - class generalization
  - Subclass hypertext specific classes from generic framework classes - class specialization
Separate the View and Data-model classes. The domain-specific view classes inherently depend upon the underlying framework, but data-model classes can be generalized for any framework.

To achieve reuse using a plug mechanism, some of the application-specific classes are designed and implemented from scratch, although their versions are available in frameworks.

- The design of a system does not stop at implementation, but the class hierarchy may be modified to improve reuse. This approach supports the object-oriented paradigm's notion of iterative-design.

We can assess that the object-oriented paradigm is a good paradigm for software reuse, although this is heavily dependent upon the approach used in design and the techniques employed in implementation. The frameworks based on this paradigm do provide classes for reuse. The technique of designing an application-specific cluster of pluggable classes can augment the reuse. On the basis of the classes presented in this thesis, we can say that we have been able to provide such generic clusters for the hypertext program. The evaluation of these classes indicates that they can be modified and maintained with changing requirements.

From a wider perspective, the current increase in the popularity of object-orientation, with its purported benefits, does not seem to be slowing down. New tools and libraries based on the object paradigm are entering the market rapidly. The advantage of using these class libraries are many, yet the important aspect should be to temper them by recognizing the degree of reuse possible. Application classes can be designed using the plug-compatible approach and can be plugged with different libraries. The short term pay-off of using such a technique may not be high in terms of investment, but the long term benefit will certainly be high.
Bibliography


