Digital Ink Annotation within Integrated Development Environments

Xiaofan Chen

This thesis is submitted in fulfillment of the requirements for the degree of Master of Science in Computer Science, completed at the University of Auckland

February 2008

Abstract

Readers are accustomed to annotating on paper documents while reading. To cater to the needs of readers, researchers have managed to reproduce this annotation ability for readers working with digital documents. However, most research has focused on textual documents rather than program code documents, and existing programming environments also do not support digital ink annotation. Therefore, this research focuses on the needs of annotating program code directly inside an Integrated Development Environment (IDE).

An IDE is a good platform for reviewing, editing, and running program code. To provide IDE with the annotation functionality, this research designed and developed a tool, CodeAnnotator, which integrates digital ink annotation support inside an IDE. This tool supports direct annotation of program code with digital ink in the IDE. CodeAnnotator enables users to make free-form annotations on program code documents without restriction of location and content, and easily modify existing annotations. This tool also supports grouping ink strokes and anchoring each annotation to a specific piece of code, thus maintaining consistency between grouped annotations and their associated code when the underlying code changes. Furthermore, this tool enables users to recognize handwritten comments, and provides a navigation system to assist users in easily and effectively reviewing and locating specific annotations. CodeAnnotator lets users enjoy both annotation and IDE supports.

To evaluate the effectiveness of CodeAnnotator, a usability study was conducted. This study obtained positive results and showed that participants were satisfied with the implementation of this tool. This study also explored weaknesses of this tool, and generated solutions or improvement ideas for each weakness. The overall feedback from participants was that this tool performed exceptionally well and was both helpful and useful to review and annotate program code.

Acknowledgements

First and foremost I want to strongly express my thanks and deepest gratitude to my supervisor, Dr. Beryl Plimmer.

Thank you to my husband and friends who have encouraged and supported me during this journey.

Finally, thanks to those who evaluated my tool, Samuel Hsiao-Heng Chang, Meghavi Doshi, Nilanthi Seneviratne, James Milburn, Paul Schmieder, Yun-Bum Kim, and Christof Cutterdh

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Have you ever borrowed a library book that contained ‘illegal’ annotations? Have you ever lamented the wastage of plastic projector handouts owing to annotations rendering them unrecyclable? Readers often underline and highlight important and interesting sections of text, or write notes in the margins when reading paper documents. This habit helps readers to understand and recall key content in the material they are reading. Comments added by previous readers can sometimes help subsequent readers understand better and read more effectively. Nowadays, documents are increasingly composed, stored and filed electronically, and readers are becoming increasingly accustomed to reading on screens and monitors. However, it remains the case that readers like to make comments and highlight sections of what they are reading. Researchers have been striving to develop systems by which readers can make annotations while reading digital documents. Unfortunately, most of these systems focus on textual format documents (e.g. Word, Txt, Pdf) rather than program code documents. However, this research provides users with a system enabling them to review and annotate program code within an Integrated Development Environment (IDE) such as Eclipse.

This chapter starts with describing the research background from which this research derived its focus. The motivation and goals of this research are then presented. Subsequently, the approach employed to fulfill the objectives of this research is described. Finally, the organization of this thesis is summarized.

1.1 Research background

An annotation is extra information (such as a marking, note or comment) made on a document at a particular place (Brush et al., 2001). According to this definition annotation is additional information added to the original source material. Adler et al. (1998) reported that annotation and note taking consume 48% of reading time. Readers augment, filter, underline, highlight, comment on, summarize and organize information as they read. This is called “active reading”

(Schilit et al., 1998). The activities involved in active reading assist readers in efficiently and effectively comprehending the reading material. Many studies have examined the influence of annotation on readers.

From the perspective of annotators, Marshall (1997; 1998) found that annotation significantly influences deep reading, serves as a bridge between reading and writing, and reflects reader engagement with the reading material. Other researchers have found that readers annotate to enhance reading-comprehension activities, assist critical thinking, aid the interpretation of the source, comment about the quality of the material, and record the reader's immediate and unselfconscious reactions to it (Bretzing & Kulhavy, 1981; Oostendorp, 1996; Marshall, Price, Golovchinsky & Schilit 1999; Ovsiannikov et al., 1999). Studies based on the perspective of subsequent readers of annotations have concluded that reading documents that have been annotated by previous readers improves subsequent readers' understanding and recall of the items emphasized by annotations (Fowler & Barker, 1974; Schumacher & Nash, 1991; Wolf, 2000). These annotations influence perceptions of specific arguments in the source material, and decrease the tendency of subsequent readers to summarize unnecessarily (Wolf, 2000). These researchers note that annotation helps not only annotators but also subsequent readers to speedily understand and comprehend the source and to memorize and recall the emphasized items.

Traditionally, readers are accustomed to making annotations with a pen on physical paper documents. Pen and paper offers two key advantages. The first advantage is that it is a natural way for readers to read and annotate paper documents that does not require any special skills (e.g. computer skills, skills in using annotation tool) and facilities (e.g. computers or software). The second advantage is the ease with which readers can sketch unstructured notes and drawings in response to the document content (Bargeron & Moscovich, 2003). However, pen and paper also has disadvantages, annotations made on paper documents are inflexible and are often disposed of, ending up in the rubbish bin.

Compared with annotations made on paper, digital ink annotations, annotations are made on digital documents using computers that allow pen-based input. Digital annotations have the advantages of enduring throughout the lifetime of a document, lending themselves to easy filtering and organization, and being easily shared just like digital documents (Bargeron & Moscovich, 2003). Moreover, continuing developments in computing and information

technology and annotation tools make it increasingly easy for readers to sketch whatever they wish on digital documents (Golovchinsky & Denoue, 2002; Bargeron & Moscovich, 2003; Olsen et al., 2004; Plimmer & Mason, 2006; Priest & Plimmer, 2006).

Most existing research has focused on digital textual documents such as essays and course notes, not digital program code documents. There has been little research on supporting annotation in digital program code files and providing digital ink annotation functionality within Integrated Development Environments (IDE). Program code documents have a unique feature: code is non-linear, being arranged in logical classes and procedures that are not intended to be read sequentially like a book (Priest & Plimmer, 2006). The non-linear structure of program code makes it inconvenient and unnatural to print it out for review and annotation. Furthermore, IDEs do not support digital ink annotation. That is, annotations cannot be made directly inside the IDEs without additional annotation support. Therefore, the focus of this research is on annotating program code directly inside an IDE.

1.2 Motivation and goals

This research aims to provide users with an electronic environment for annotating program code documents within an IDE, and is motivated by the following three considerations.

First, the earth has finite resources. Producing paper consumes resources and damages the environment. Also the limitations of paper documents make it difficult to manage the transfer, storage, modification, and preservation throughout the document's lifetime. A paperless environment (i.e. digital environment) frees readers from the location and physical constraints of paper, provides better support for updating, filing, sharing and comprehensive searching of documents, and preserves the digital documents throughout their lifetime (Plimmer & Apperley, 2007).

Second, Colen (2001) indicated that performing a thorough review of program code is one of the best approaches for significantly enhancing software quality. Reviewing and annotating program code on paper suffers from two key disadvantages. First, it is inconvenient, unnatural and time-consuming to annotate printed copies of program code documents because the non-linear form of code makes it impossible to read like a book; readers must manually search

through numerous pages to find variables, methods and classes (Priest, 2006). Second, reviewing annotations made by others is extremely time-consuming because of the need to manually search through numerous pages to identify where annotations are made. Digital ink annotation over digital source code overcomes the above two drawbacks.

Third, Ben-Ari (2001) observed that when learning to program, it is essential that students are given the opportunity to practice in an environment where they can receive constructive and corrective feedback and practice how to write, compile and execute program code. Feedback is the most effective and efficient way to actively learning. However, IDEs do not support digital ink annotation, making it difficult for markers and teachers to add feedback to digital program code files, and also for students to program, compile and execute code while reviewing feedback.

The goals of this research are:

- To investigate how to integrate digital ink annotation functionality into an IDE,
- To conceptualize, design and develop a prototype digital ink annotation system for program code documents to support users annotating program code within the IDE,
- To evaluate the effectiveness of directly annotating over digital program code files inside the IDE.

1.3 Approach

To fulfill the above research goals, this thesis first conducts a wide-ranging literature review. Specifically, the literature review focuses on traditional and digital ink annotation and the technical challenges involved in accomplishing successful digital ink annotation. Next, the literature review examines existing digital ink annotation systems to explore valuable ideas that can be incorporated into a program code digital ink annotation system within an IDE.

Based on the results of the literature review, a conceptual plan of a program code digital ink annotation system is carried out. The functions the system needs to support and the issues need to be overcome are identified. The approach used to integrate the system into an IDE is also investigated. Additionally, the hardware and software requirements of the system are analyzed.

Next, according to the conceptualization of a program code digital ink annotation system, this thesis designs and develops a tool, CodeAnnotator, to support users directly annotating over program code within an IDE. With CodeAnnotator after loading a code file in the IDE, users can write annotations using a digital pen while also having all the functions offered by the IDE, such as debugging, compiling, and executing. Therefore, users have both IDE and annotation support.

Finally, one in-depth evaluation is conducted to examine the strengths and weaknesses of CodeAnnotator. The evaluation is conducted from the viewpoint of users, including annotators and annotation reviewers, to gather feedback on usability and extra ideas about the tool.

1.4 Thesis overview

This thesis is organized as follows:

Chapter 1 introduces the background to this research, its motivation and goals, and the approach used to fulfill those goals.

Chapter 2 reviews traditional and digital ink annotation to explore technical challenges involved in accomplishing successful digital ink annotation and existing digital ink annotation systems to investigate valuable ideas that a program code digital ink annotation system needs to support.

Chapter 3 describes the requirements and functions of a program code digital ink annotation system that is integrated into an IDE.

Chapter 4 describes the implementation of CodeAnnotator developed in this research and issues encountered during its design and development.

Chapter 5 evaluates CodeAnnotator by examining its strengths and weaknesses from the perspective of annotators and annotation reviewers.

Chapter 6 discusses the technical aspects, implementation and evaluation of CodeAnnotator. The evaluation includes the strengths, weaknesses and possible improvements of CodeAnnotator.

Chapter 7 presents conclusions and contributions of this research and future work of CodeAnnotator.

1.5 Summary

This chapter introduced this thesis topic and the area this research focuses on, as well as presenting the research motivation and goals, and outlining the proposed approach. The next section reviews concepts and ideas related to annotation, challenges in supporting digital annotation, and existing digital ink annotation systems.

Chapter 2

Related Work

This chapter presents a literature review. Annotating over documents while readers are reading is a common technique used to support active reading. This research is interested in annotating directly over program code inside an IDE and integrating digital ink annotation support within the IDE. To achieve this, it is necessary to first review related research on annotation to explore how annotations are made, the types of annotations made, the functions of annotations, the popularity of various kinds of annotations, and the effectiveness of digital ink annotation on digital files. These issues are presented in the "Traditional ink annotation" section, which also details the advantages and disadvantages physical ink annotation has.

Section 2.2 discusses related work on digital ink annotation to explore the requirements of hardware and software involved in accomplishing successful digital ink annotation, along with the advantages and disadvantages digital ink annotation has. In section 2.3, various existing digital ink annotation systems are discussed to seek valuable ideas in achieving a successful program code digital ink annotation tool. Finally a summary is presented.

2.1 Traditional ink annotation

Paper has become seamlessly integrated into people's daily life, study and work. Although exponential improvements in information technology (e.g. Computer, Internet, Telephone, and Mobile) have enabled people to live and work in a paperless environment, paper still continues to play a significant role. Readers continue to prefer to read documents on paper. The preference for paper also applies when annotating documents. Annotating paper documents using pens is considered as a traditional pen and paper approach. Annotation over paper documents is called traditional (or physical) ink annotation.

How do readers make annotations over documents? To make annotations, readers require a paper document and writing tools such as pencil, colored pen, and colored highlighter.

Functions vary among different writing tools. Annotations made by pencil can be readily and easily removed with an eraser without leaving any trace. Pen can be used to write legible marginal notes or underline passages. Highlighter can be used to highlight text or draw marginal bars. The behavior of readers making annotations exhibits three notable characteristics (Marshall, 1997). First, readers using highlighters write fewer marginal notes than those underlining passages using pens, because it is far more difficult to write legibly with a highlighter than with a pen. Second, annotators develop complex coding schemes, such as highlighting different passages with different highlighter colors to signify different types of information. Finally, forms of annotation are shaped by disciplinary expectations and textbook genre. For example, pencil is the preferred annotating tool in mathematics, complex philosophical narratives are normally highlighted and underlined, and difficult works typically teem with marginal notes and jottings. These annotation characteristics indicate that annotators vary their annotation strategy (for example pencil versus colored pens or highlighters) depending on the content they are reading. The question thus arises of how many different types of annotations exist?

Marshall (1997) classified annotations into three main types: annotations within the text, annotations in margins or blank spaces, and removable annotations. Annotations within the text include underlining, highlighting, and circles and boxes around words and phrases, and brief notes written between lines. Annotations in margins or blank spaces include explicitly textual notes or jottings, marginal symbols (e.g. brackets, angle brackets, braces, asterisks, and stars), circles and boxes around whole pages, and arrows and other symbols linking marginal markings. These first two types of annotations are written directly on paper documents with pencil or colored pens or colored highlighter. In contrast, removable annotations are not written on the paper itself and are easily removable. Such annotations are made by removable media such as bookmarks, dogears, and Post-its, or using pieces of note paper tucked into the pages. However, this research only focuses on annotations within the text and in margins or blank spaces, because this research is interested on annotations on documents not on removable media, and also program code documents normally contain a large amount of blank space that notes and comments could easily occupy (Priest, 2006). These two types of annotations allow users to make any kind of symbols, words and text selection marks at anywhere inside the page. Then what benefits do readers want to obtain from annotations? Or what functions do annotations have?

Making annotations on documents helps readers to interpret, comprehend, organize, and summarize the information, and to highlight, memorize, and recall important information. Marshall (1997) found that annotations serve numerous functions. She enumerated six most evident functions.

- First, readers use annotations to facilitate future rereading of the underlined and highlighted sections.
- Second, annotations (including short highlights, circled words or phrases, marginal markings like asterisks) serve as place-markings and memory aids for assisting future recall.
- Third, extended comments written near problems or equations or theorems can analyze, answer or explain problems, equations or theorems.
- Fourth, marginal notes and notes between lines and jottings are used as a record of interpretation or opinions of nearby texts.
- Fifth, extended highlighting and underlining can act as a visible trace of the reader's attention through difficult narrative.
- Finally, markings (such as notes, doodlings and drawings) unrelated to the material themselves reflect circumstances entirely outside of the realm of the material.

These six functions indicate that annotators use different kinds of annotations to obtain different benefits. While every kind of annotation has specific functions and is useful in specific circumstances, an important question is which kinds of annotations annotators prefer to use?

Researchers have found that the majority of annotations are highlights and underlines (O'Hara & Sellen, 1997; Marshall & Brush, 2002; Marshall, 2003; Bargeron & Moscovich, 2003; Shipman et al., 2003). O'Hara and Sellen (1997) noted that most of annotations are simply underlines and highlights without words. This observation was reinforced by Marshall (2003) who examined the popularity of different kinds of annotations. She found that only 28% of annotations are words and symbols, with remainder (72%) merely comprising underlines, highlights and circles. Bargeron and Moscovich (2003) similarly found that 76% of annotations are simply underlines, highlights and circles, with notes comprising only 24%. These figures indicate that majority annotations simply comprise underlines, highlights, and circles. However, whether this means that notes and symbols are unimportant remains to be determined?

In most circumstances, a document is reviewed by multiple individuals; annotators review documents and make annotations, then the annotated documents pass on to others for further review and annotation. For example, a teacher or marker may read and annotate the assignment of a student, and the annotated assignment may then be returned to the student for review. The issue of whether annotations can be shared between readers thus arises. Marshall and Brush (2002; 2004) found that only annotations containing words can be shared and understood by other readers and annotations that support comments together with a point of reference facilitate enhanced understanding. Furthermore, annotations that are shared among readers also support communication and collaboration (Cadiz et al., 2000). If shared annotations only contain underlines, highlights, and circles, they become unable to facilitate effective and efficient communication and collaboration. Therefore, for shared documents notes and symbols are extremely crucial in facilitating the wide dissemination of private annotations. Since program code documents are generally shared among developers and reviewers, it is better to annotate such documents using notes and symbols.

The above literature review demonstrates that the traditional pen and paper approach enables users to make various kinds of annotations with various writing tools, and each represents specific functions, and moreover shared documents are more suitable for making annotations containing words. The pen and paper approach offers numerous advantages which have encouraged readers to continue to use this method to do annotations, as follows:

- Paper and writing tools are ubiquitous and inexpensive.
- It is natural for readers to read and annotate paper documents since doing so requires no special skills and facilities to do these activities. Readers do not require any special knowledge of computers and annotation, and nor do they have to buy a computer and annotation system.
- Annotating paper documents can be performed anywhere as long as writing tools are available, writing tools are far easier to transport than a computer.
- Annotations can be added faster and more accurately by hand than by keyboard, especially for annotations in non-alphabetical languages such as Chinese and Japanese (Hamzah et al., 2006).

- Readers can easily sketch unstructured notes and drawings in response to document content (Bargeron & Moscovich, 2003), and can make annotations anywhere on the paper.
- Paper is tangible. Physically seeing the document can act as a tangible reminder that it needs attention, and scanning through the paper document enables readers to quickly obtain an overview of its layout and content (Marshall, 2003).

The traditional pen and paper approach also has many disadvantages.

- Producing paper consumes resources and damages the environment.
- The volume of storage place required gets bigger and bigger as paper documents accumulate.
- Annotations cannot be reedited by being erased, re-colored, moved and modified. Only annotations made in pencil can be erased without trace.
- Traditional ink annotations on paper documents cannot be widely shared among readers because it is time-consuming, expensive and difficult in printing, copying, and distributing.
- It is hard to keep annotated paper documents permanently because paper documents suffer wear and tear every time they are used. Annotated paper documents thus often end up in the rubbish bin.

To emulate pen-and-paper annotation abilities on computers and digital documents, digital ink annotation systems must preserve the advantages of the pen-and-paper approach while overcoming its weaknesses. The above literature indicates that digital ink annotation systems should provide various colored pens and highlighters, support multiple types of annotations, and permit annotations to be made without constraints of space and to be shared among readers. Before examining the existing digital ink annotation systems, this thesis examines the determinants of the decision of readers to review and annotate digital documents.

2.2 Digital ink annotation

For centuries readers have been accustomed to writing annotations with pencils, pens or highlighters while reading paper documents. For a long time paper was the only means of

presenting documents, and pen and paper was the only method of making annotations (Crawford, 1998). However, the advent of computer technology has made it possible to edit and read electronic documents. Nevertheless until the middle of the 1990s, widespread reading and editing of electronic documents was hindered by the inherent characteristics of early computers, including: it was high cost; operating systems that required users to be highly proficient in using computers; available hardware and software were poorly suited for performing complex tasks such as editing audio and video. These problems were alleviated when Microsoft released Windows 95 in 1995, which was a user friendly consumer-oriented graphical user interface-based operating system (Windows History, 2002). Simultaneously, falling computer prices further stimulated the more widespread adoption of computers. Computers thus gradually became part of everyday life, a trend further stimulated by the popularity of the Internet and the continued evolution of hardware and operating systems (e.g. the evolution from Microsoft Windows 95 to Vista). Readers have now become accustomed to performing many routine tasks on computers, including editing documents or images, searching for information on the Internet, making daily plans etc. Marshall (2003) indicated that "much of what we read when we research a topic now arrives in digital form". This reflects that most documents are edited on computers and stored electronically. However, despite the enormous growth in the use of computers during the past couple of decades, whether readers prefer reading documents on computers remains unclear.

In reality, readers still tend to read electronic documents as printouts where possible. O'Hara and Sellen (1997) studied this phenomenon and observed that the benefits of paper far outweigh those offered by computers when choosing to read a document. However, readers are gradually developing the habit of reading documents on computers in response to improvements in computer screen technology and computing technology (e.g. portable computers, hardware, operating systems, and advanced reading software). These improvements enable users to read electronic documents naturally and intuitively. Readers are becoming used to reading digital documents on computers, but whether they are also becoming used to annotating digital documents remains uncertain.

Annotating documents as readers read is a natural and intuitive way to carry out active reading. Readers often use pen and paper when annotating documents because of the advantages of this approach (O'Hara & Sellen, 1997; Schilit et al., 1998). Annotating and reading should be interleaved seamlessly, but in the computer context, reading was interspersed with long periods

of typing and comments were input with keyboard and mouse or a graphic tablet with a pen (See Fig. 2.1a). This was because readers had to keep one eye on the screen and another on the keyboard or tablet (O'Hara & Sellen, 1997). However, since 2002, the advances in improvements in hardware, operating systems and software have enabled users to enjoy reading and annotating digital documents on the computer.

a **b**

Figure 2.1: Tablet devices. (a) Graphical Tablet with Pen. (b) Tablet PC

(**a** was retrieved from http://www.gpstore.co.nz/Hardware/1468436.html; **b** was retrieved from http://en.wikipedia.org/wiki/Tablet_PC)

In 2002, Microsoft released the first Windows XP Tablet PC Edition. Windows XP Tablet PC Edition was subsequently upgraded in 2005, realizing the long-held industry vision of mainstream pen-based computing (Windows History, 2002). This operating system allows users to manipulate and process handwritten data. The system runs on a Tablet PC (See Figure 2.1b), a specially designed notebook computer that incorporating two devices not found in conventional computers: a digital pen for handwriting and a pen-sensitive screen which can be rotated 180° and folded down over the keyboard to offer a flat writing surface. The Tablet PC allows users to input data by "inking" with a digital pen using "digital ink", as well as inputting data via a standard keyboard or mouse (Redmond, 2002). The combination of the specialized operating system and the Tablet PC enables users to read and annotate digital documents naturally, intuitively, conveniently and effortlessly. A further two characteristics provide additional incentive for users to perform reading and annotating activities on computers. First, Tablet PCs have become increasingly affordable and widespread owing to recent progress in computer technology and the reduction of manufacturing costs. Second, many systems have been developed to help users make annotations on digital files. Some existing digital ink annotation tools are described in the next section.

Nowadays, readers are increasingly willing to read and annotate digital documents on computers because of the advantages offered by the Tablet PC and the Tablet operating system, digital ink annotation tools and computing technology. Digital ink annotation has the following advantages:

- It encourages readers to adopt paperless work habits, thus reducing paper use.
- The unique feature of the Tablet PC (its touch-sensitive screen which can be rotated and folded back over the keyboard to provide a flat reading and writing surface) allows users to read and write naturally and intuitively, similar to reading a paper document and writing on it with a pen.
- Digital ink annotations on a document can easily be stored, searched, shared and transferred via the Internet by using the file system of the operating system (Priest, 2006). Stored and shared digital ink annotations can be reviewed by anyone regardless of spatial constraints.
- Digital ink annotations on a document can be readily and easily modified, recoloured, moved, added, and deleted without leaving the untidy traces that often happen accompany such modifications in traditional ink annotations. In paper documents modification of annotations typically involves concealing or crossing out the annotations and then rewriting them.
- Readers can review or edit digital ink annotations anywhere as long as they can access the computer. Furthermore, they do not need to worry about their annotations being lost.
- Digital annotation tools provide various pens and highlighters with different colors and thicknesses. Users do not need to worry about running out of ink.
- In the education area, student attention and comprehension during class are enhanced when the Tablet PC together with a digital ink annotation tool are used to present teaching materials. Students prefer this teaching approach to traditional black- or whiteboard teaching (Anderson et al., 2004; Mock, 2004)

However, digital ink annotation has one major disadvantage:

- Without Tablet PC and digital ink annotation tools, it is difficult and inconvenient to make annotations on digital documents. Additionally, compared to paper annotation, which is effortless and smoothly integrated into reading, digital ink annotation on digital

files is relatively cumbersome and distracts from the reading task (O'Hara & Sellen, 1997).

2.3 Digital ink annotation systems

To provide a natural and intuitive method of making annotations on digital documents, numerous digital ink annotation systems (tools) have been developed for documents with different formats, including text files, Microsoft Office files, PDF, Image files, Code files and so on. These tools have inherited the advantages of the pen-and-paper approach while also offering additional functions to help users to easily and conveniently make and review annotations. Different tools provide different features and support documents in different formats. Figure 2.2 compares existing digital ink annotation tools in detail.

Annotation tool name	Integrated into an environment	Supporting multi-format documents	Capable of editing document content	Free-form annotations	Grouping annotations	Anchoring annotations	Reflowing annotations	Supporting navigation system
XLibris	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Text Annotation Software (TAS)	Yes	No	Yes	No	No	Yes	Yes	Yes
Microsoft Office products	No	No	Yes	Yes	No	Yes	Yes	No
Callisto digital ink annotation	Yes	No	Yes	Yes	Yes	Yes	Yes	No
Electronic Student Notebook project	No	Yes	No	Yes	No	No	No	No
Ink Annotations on Web Documents (IAWD)	Yes	No	No	Yes	No	Yes	No	No
ScreenCrayons	No	Yes	No	Yes	No	Yes	No	No
Penmarked	No	Yes	No	Yes	No	No	No	No
Rich Code Annotation Tool (RCA)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No

Table 2.1: Comparison among ten current digital ink annotation tools

Integrated into an environment

"Integrated into an environment" means the annotation tool is integrated into a system that has the extension capability such as Eclipse, Visual Studio, and Internet Explorer etc. Table 2.1 above shows that most of tools (5 out of 9) are self-reliant and independent from other

environments. Only four tools are integrated with other environments, and these environments have the support of digital ink annotation. The four tools are described in detail below:

- Text Annotation Software (TAS) (Brush et al., 2001) is an extension of Microsoft Internet Explorer. TAS allows users to highlight and make text notes on web pages (See Figure 2.2). Users can highlight or attach a note to a portion of text by selecting a portion of text with the mouse and then left-clicking the selection. Highlighted text is displayed in yellow and text with an attached note is displayed in blue. Annotations for the web page are displayed in the annotation index window. This tool enables the repositioning of existing annotations while the web page is being modified.

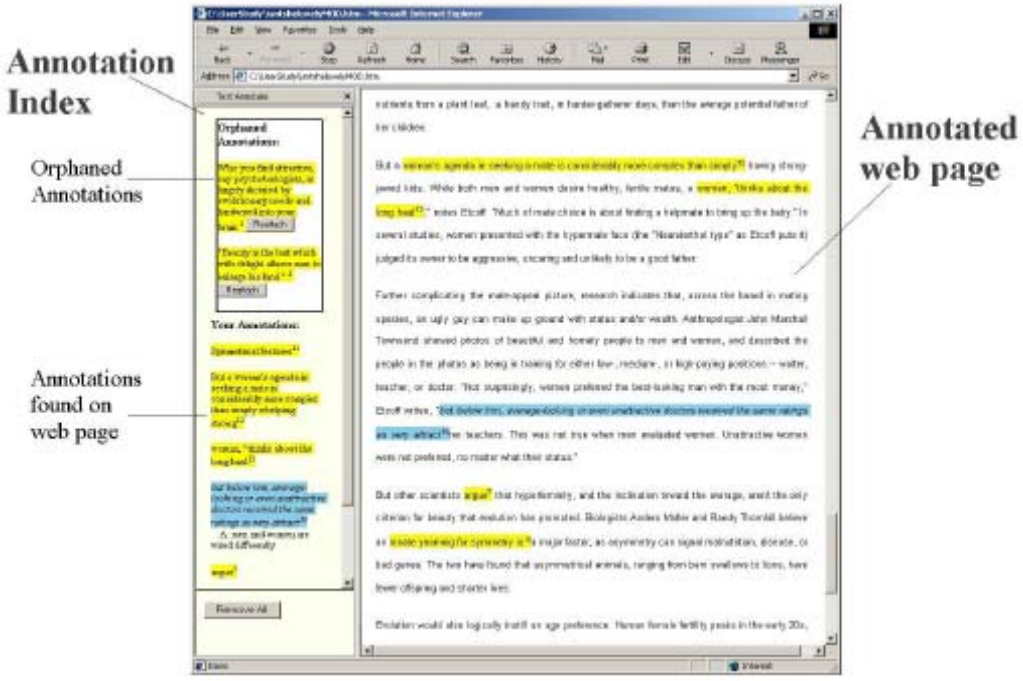


Figure 2.2: Text Annotation Software

- Callisto digital ink annotation (Bargeron & Moscovich, 2003) is a plug-in for Microsoft Internet Explorer (See Figure 2.3). Callisto is based on their own framework that supports an IE toolbar with a pen and a highlighter that allows users to mark any part of a web page with digital ink. This tool groups and classifies annotations, and then anchors grouped annotations to a context within the document. This ensures that annotations move to compensate for any changes in the layout of the original document.

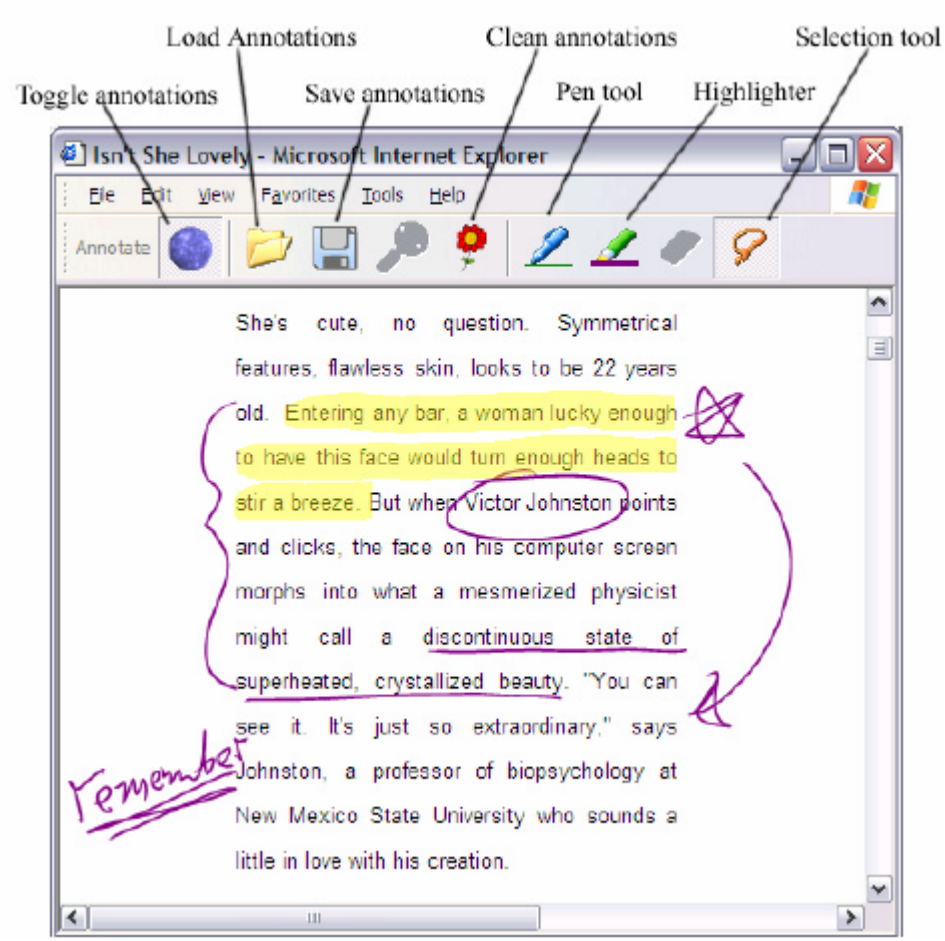


Figure 2.3: Callisto digital ink annotation

- Ink Annotations on Web Documents (IAWD) (Ramachandran & Kashi, 2003) is an annotation tool that is integrated into web browsers (See Figure 2.4). IAWD first captures, then renders and associates digital ink annotations with the specific underlying text elements of a web page using W3C's Document Object Model (DOM) and Dynamic HTML (DHTML).

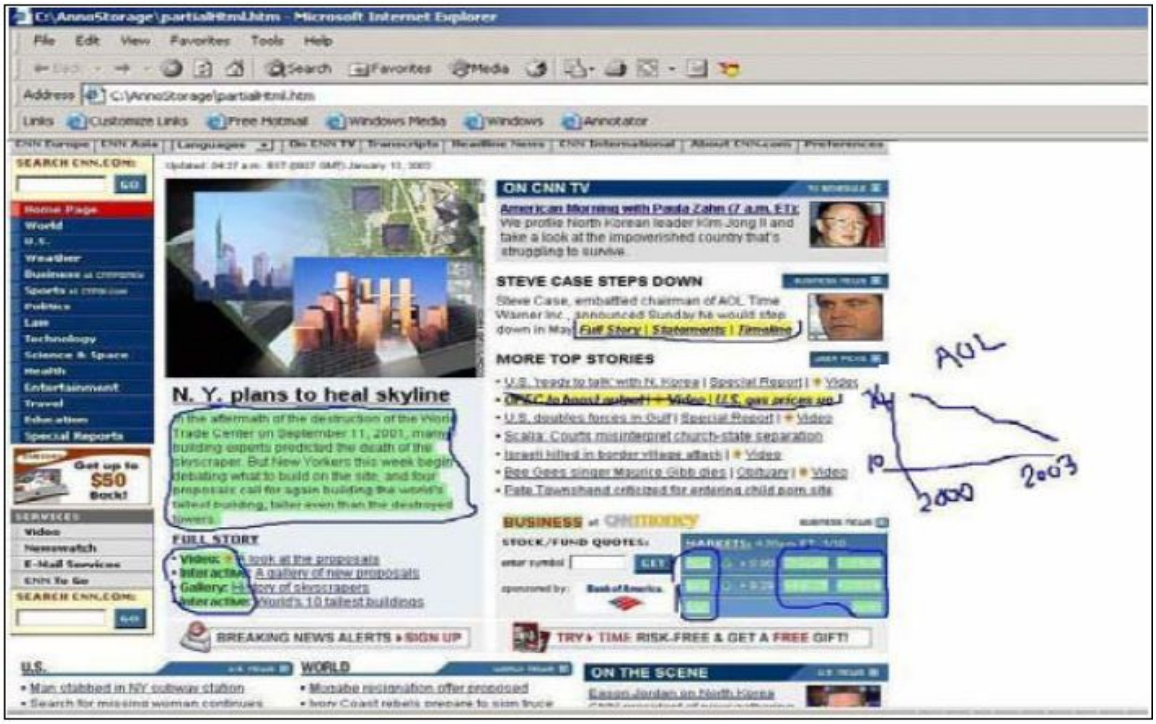


Figure 2.4: Ink Annotations on Web Documents

- Priest and Plimmer (2006) have developed Rich Code Annotation Tool (RCA), which is a plug-in for Microsoft Visual Studio .NET 2005 IDE (VS) (See Figure 2.5). RCA is implemented in C# using both the Microsoft .NET extensibility model and the Microsoft Ink API. Users make annotations on a program code document in the ink window and edit the code in the code window. Users not only enjoy annotation support but can also use all the functions provided by VS. Furthermore, Priest and Plimmer (2006) reported significant difficulty in seamlessly integrating the annotation function into VS. They explored three approaches to hold annotations: the first approach involved building a transparent overlay directly over the code window to hold annotations; the second approach involved creating a new associated ink window that was tightly-coupled to its code window much like an associated 'design' window for a user form in a .Net windows application; the third approach involved building a separate and distinct ink window. The first two approaches were difficult to implement due to the limitations of the extensibility model. Therefore it was necessary to compromise and employ the third approach to hold ink annotations. This tool indicates the difficulty of seamlessly integrating an annotation tool into an IDE.

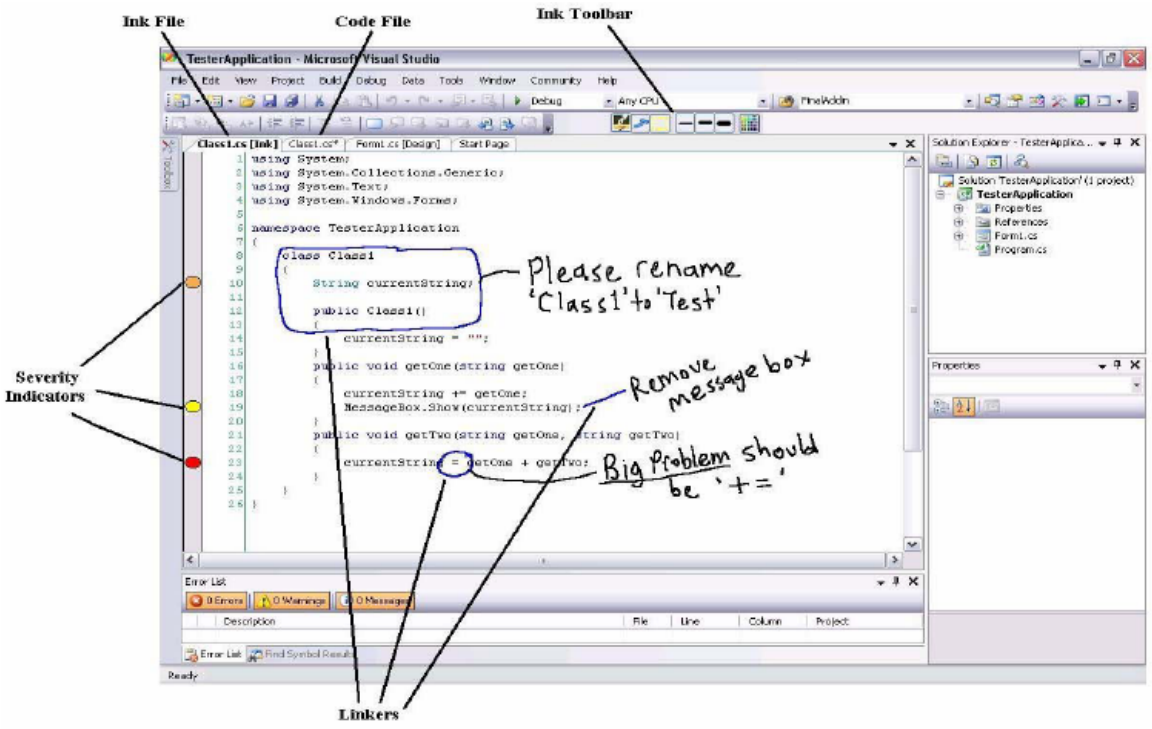


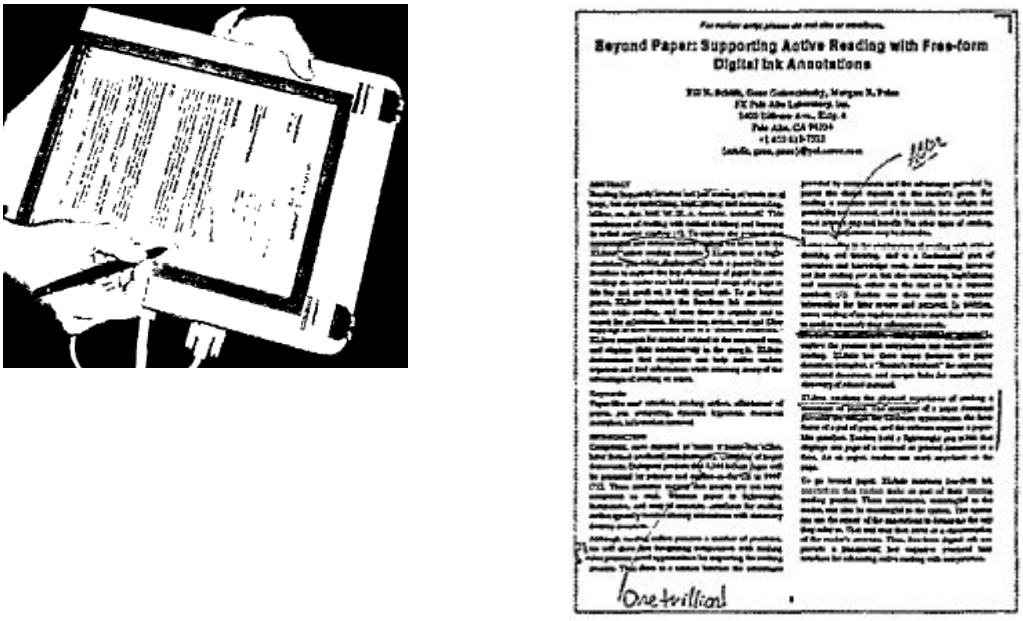
Figure 2.5: Rich Code Annotation Tool

Three of the above four tools are integrated into web browsers. Only RCA is an extension of an IDE. Moreover, RCA demonstrated the difficulty of making annotations directly over the code window inside an IDE.

Supporting multi-format documents

Digital documents are stored in various formats. Normally, a specific application can only support certain types of documents; for example, Microsoft Excel only supports Excel files, Visual studio supports most programming languages (VB, C++, C#, J#) and so on. Most annotation tools only support annotations in specific format documents, such as Microsoft Word supports text and Word files, Microsoft Excel only supports Excel files, Microsoft Powerpoint and Class Presenter (Anderson et al., 2004) only support Powerpoint files, Collaborative Annotations on Visualizations (Ellis & Groth, 2004) only supports image files, and TAS, Callisto, IAWD, WPM (Koga et al., 2005) and MADCOW (Bottoni et al., 2006) only support HTML web pages. Creating a pervasive annotation tool is an architectural challenge. Five approaches are used to support annotations on multi-format documents.

One approach to annotation used by certain annotation tools is to create a special purpose model for all documents to be annotated (Olsen et al., 2004). For instance, XLibris (Schilit et al., 1998) uses a document-centric approach and an “image plus text” file format (Phelps & Wilensky, 1996) to support multi-format documents (See Figure 2.6). The disadvantage of this approach is that the model restricts what can and cannot be annotated because the model is unable to represent all kinds of documents (Olsen et al., 2004). Another drawback is that this approach treats digital documents the same as printed-out documents, preventing changes from being made and only allowing annotations.



a **b**

Figure 2.6: XLibris (1998). It was created in 1998, (a) Tablet display, (b) an annotated document in XLibris

The second approach is to annotate exclusively in image space, to render the information contained in documents as images using the ability of Window systems to capture screen images (Olsen et al., 2004). This approach enables users to annotate any information from any application without requiring the cooperation of that application (Olsen et al., 2004). For instance, ScreenCrayons (Olsen et al., 2004) employs this approach to provide a universal annotation facility and infers simple structure from the image itself to exploit the structure of the document to behave intelligently (See Figure 2.7). The weakness of this approach is that it is easy to lose the structure of the annotated document. Another disadvantage is losing some document information for example some of the non-visible context.

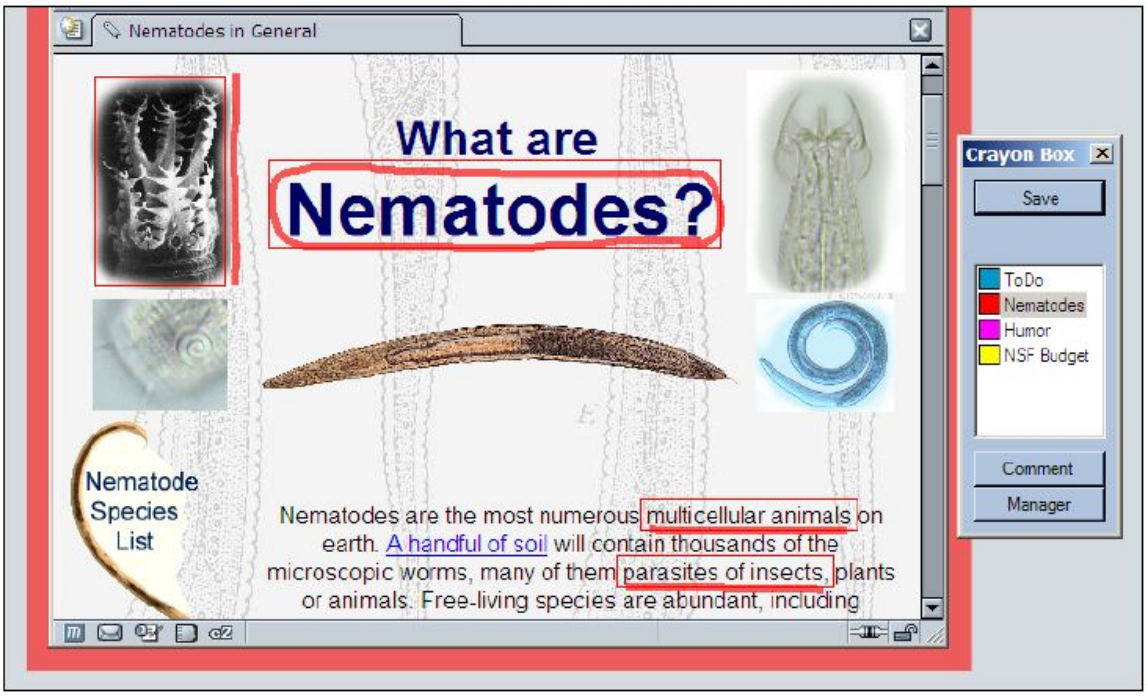


Figure 2.7: ScreenCrayons

The third approach is to export documents into Windows Metafile (WMF) images and used as an image underlay on which users can make annotations (Huang et al., 2003). For instance, the Electronic Student Notebook project (Huang et al., 2003) is implemented using C# and Microsoft .NET framework. This project uses this approach to support multi-type documents, but currently only supports the annotation of Microsoft Word and Powerpoint documents, with support for PDF files planned in the near future. Its user interface (See Figure 2.8) specifies the supported document formats and defines the exact structure of the annotations and how to display them alongside the document. However, this approach does not support users editing annotated documents, which are fixed.

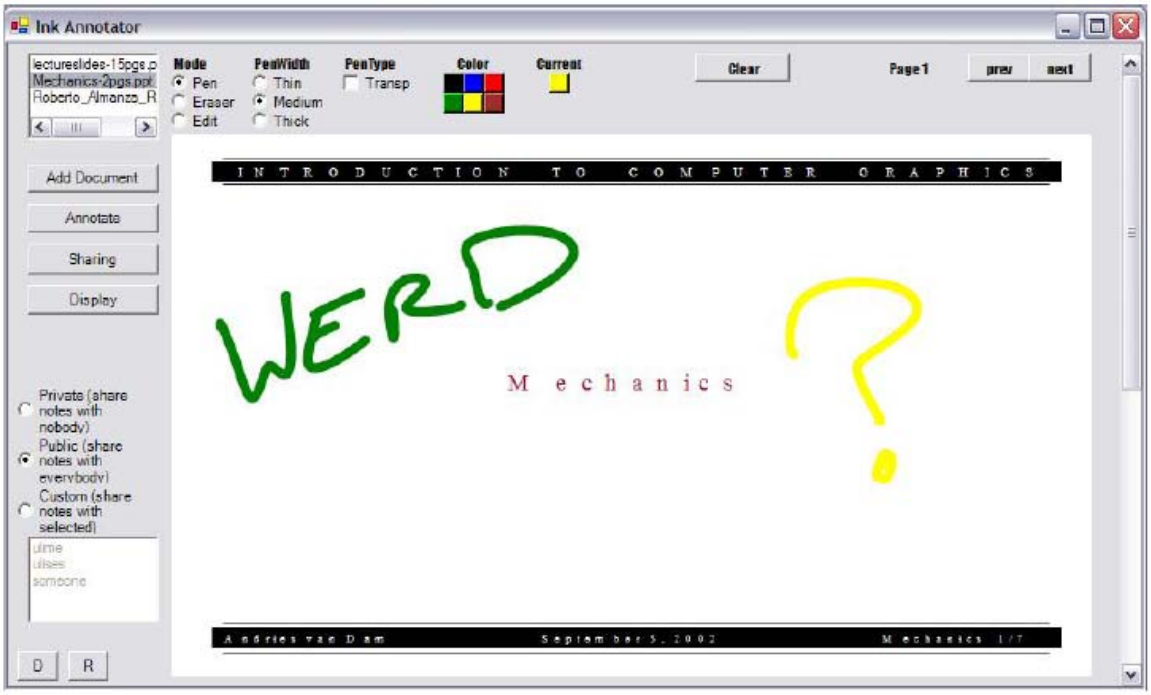
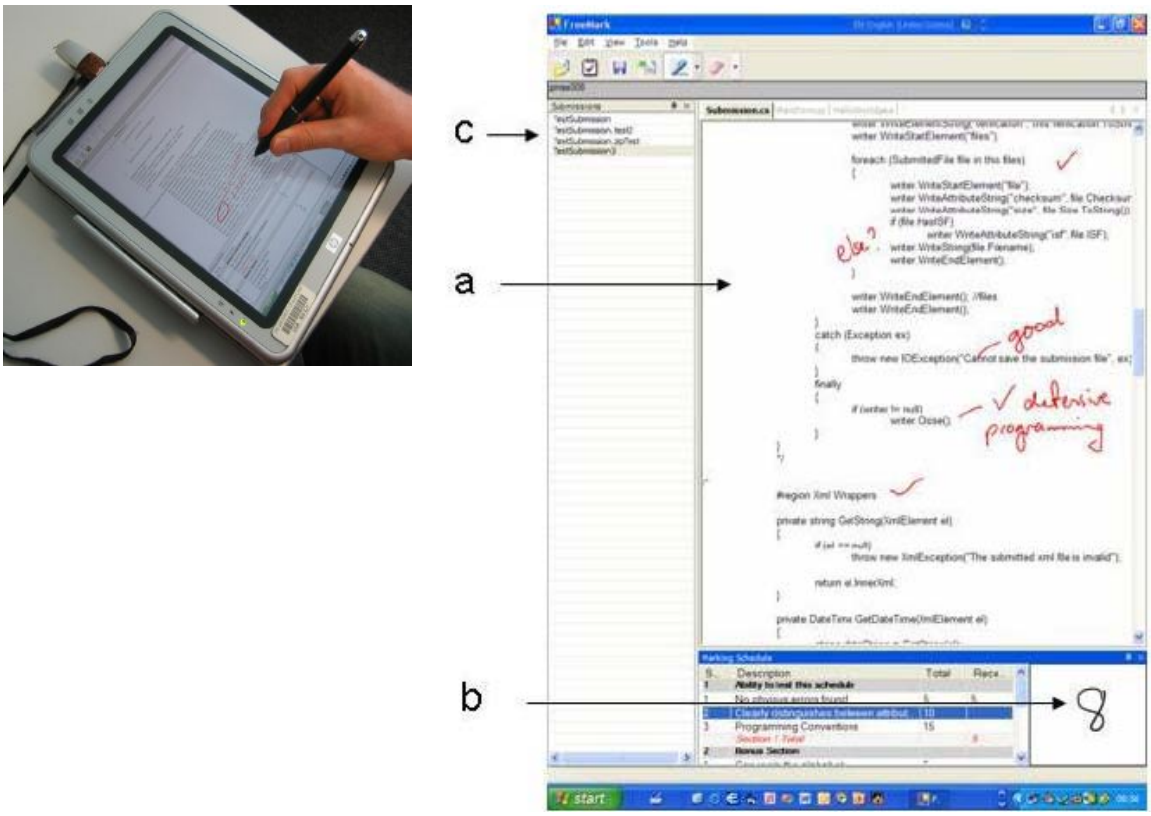


Figure 2.8: Electronic Student Notebook project user interface. Documents available for annotations appear in the top left. Users can select sharing models in the bottom left and select a pen's width or color in the top.

The fourth approach is to develop an annotation tool to support specific kinds of documents or to integrate the annotation tool with a system that supports multi-format documents. For example, since Visual Studio supports many programming languages and RCA is integrated with Visual Studio, RCA (Priest & Plimmer, 2006) annotate documents written in these programming languages. This approach only permits the tool to be implemented inside an integrated environment.

The final approach involves transferring certain types of documents into textual format files and developing an annotation tool to support those text files. For example, Penmarked (Plimmer & Mason, 2006) used such an approach to support multi-programming languages. Penmarked is a tool for annotating and marking student assignments on Tablet PCs. Teachers can read the assignments and add digital ink annotations directly on the document. (See Figure 2.9) However, in this approach some document information is discarded; for example, the code files are affixed and cannot be debugged or executed.



A **B**

Figure 2.9: Penmarked. (A) Tablet interface; (B) Penmarked user interface layout including (a) annotation pane, (b) mark schedule, (c) student list

The above five approaches show that it is unreasonable and infeasible to design a unified annotation tool that supports multi-format documents because document formats are so diverse, and it is impossible to display documents without discarding information, such as losing the ability to edit, losing the document structure (Huang et al., 2003).

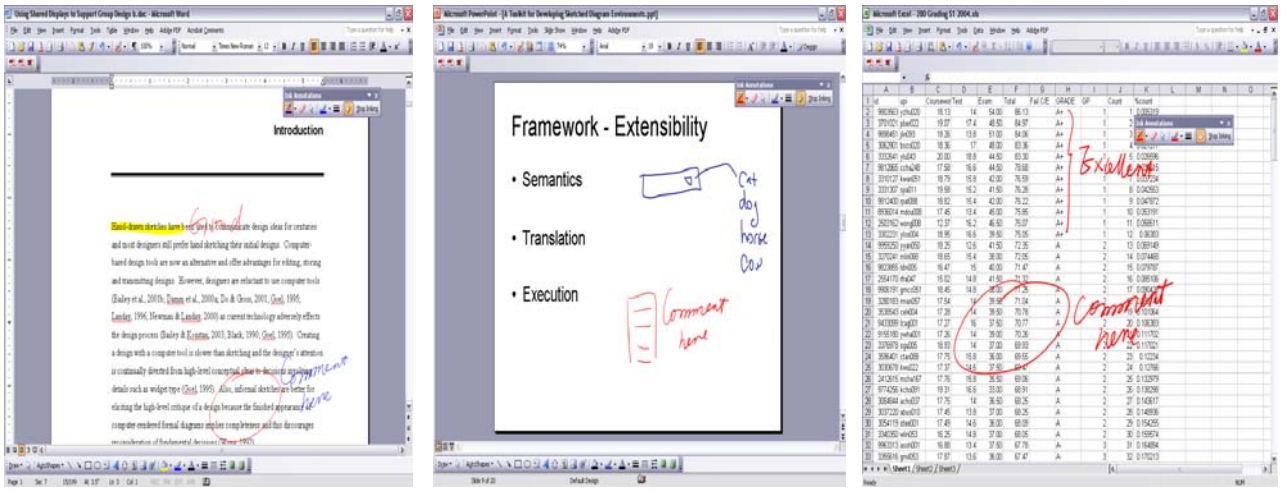
Capable of editing document content

One of the key advantages of digital documents is that their content can be readily and easily edited or modified at any time. A digital annotation tool should inherit this ability to allow users to edit annotated documents. Otherwise, there would be no difference between annotating digital and printed-out documents as they are all affixed. Five annotation tools support users in editing underlying content (Table 2.1).

- The first is XLibris: the earlier version of XLibris (Schilit et al., 1998) is treated as a substitute for a printer and supports static documents, while the later version of XLibris

(Golovchinsky & Denoue, 2002) only enables the font size of loaded documents to be changed.

- The second is TAS that allows users to modify the annotated document without losing the visual meanings and styles of the annotations.
- The third is Microsoft Office products (Microsoft Office Online, 2008). Users can edit or modify the underlying content while making annotations in Microsoft Word, Excel and Powerpoint. (See Figure 2.10)
- The fourth is Callisto. It only supports changes in document layout.
- The last one is RCA. RCA allows users to modify the annotated program code file at any time without damaging the consistency of existing annotations and their underlying contents. However, RCA maintains two copies of the code – one for annotating and one for editing. In other words, one copy of the code is contained in the ink file, while the other is contained in the code file. The copy in the ink window is immediately updated to reflect any changes of the code in the code window.



a **b** **c**

Figure 2.10: Microsoft Office products. (a) Microsoft Word, (b) Microsoft Powerpoint, (c) Microsoft Excel

Free-form annotations

Free-form annotation means that an annotation can comprise any words, symbols or marks and can be made anywhere in the document without restrictions of shape or content (Golovchinsky & Denoue, 2002). When annotating paper documents, annotators can make any kinds of

annotations such as highlight, underline, marginal notes, any forms of symbols and so on (Marshall, 1997). The factor that annotations on digital documents are as rich in form as markings on paper is one of important aspects in assessing the quality of an annotation tool (Marshall, 1997). In other words, it is essential for an annotation tool to allow users to make any types of annotations. Table 2.1 shows that there is only one tool unable to support this function, namely TAS, which only supports highlighting and making text notes.

Grouping annotations

An annotation usually consists of multiple ink strokes. Individual ink strokes must be grouped into annotations with roughly the same accuracy as a human would (Bargeron & Moscovich, 2003). For example, an annotation contains an arrow line and a comment "How?", which the annotator naturally treats as a group but a computer does not. Handwriting is characterized by the rapid creation of spatially proximate strokes (Golovchinsky & Denoue, 2002). Groupings ink strokes thus can be determined using their temporal and spatial properties (Golovchinsky & Denoue, 2002; Bargeron & Moscovich, 2003). Temporal property is the time between the current stroke being made and the last stroke being completed. If the time is less than 500 milliseconds, then the two strokes are assigned to the same group (Golovchinsky & Denoue, 2002). Spatial grouping refers to grouping strokes that are spatially close. In Callisto, users manually select a set of strokes by using the selection tool to group and classify ink strokes (Bargeron & Moscovich, 2003). RCA groups strokes based on their temporal and spatial properties, with any two strokes made within two seconds being grouped together, and users also being able to click on an existing annotation under the linker mode and any new strokes then being added to the selected annotation (Priest & Plimmer, 2006).

Anchoring annotations

From the perspective of annotators, every annotation is meaningful and associated with some specific content, like several words, a figure, a theory, a line or several lines of text etc. This content is called the anchor of the annotation (Golovchinsky & Denoue, 2002). For annotations on paper it is unnecessary to specify which content the annotation is attached to. However, when annotating digital documents it is necessary to define the location and content associated with the annotation in the computer and save this information together with annotations for the annotations to stay where they should be when they are reloaded. Bargeron and Moscovich

(2003) observed that physical position within digital documents loses meaning when documents change and annotations must be anchored to their surrounding logical context (such as the associated text). Table 2.1 lists six annotation tools that are able to anchor each annotation to its associated context.

- XLibris (Golovchinsky & Denoue, 2002) anchors annotations under the situation that the document structure remains unchanged (See Figure 2.11). This tool defines the anchor of each annotation via the following approach: the anchor of an annotation attached to a line of text is the set of words intersected by the annotation; the anchors of margin bars and circled passages are the first and last words on the lines whose bounding boxes intercept the vertical position of the annotation; for handwritten notes and symbols written in the margin, the same approach as for margin bars is applied; finally, annotations written in the upper or lower margins of a page are attached to the first of last line of text on that page (Golovchinsky & Denoue, 2002).

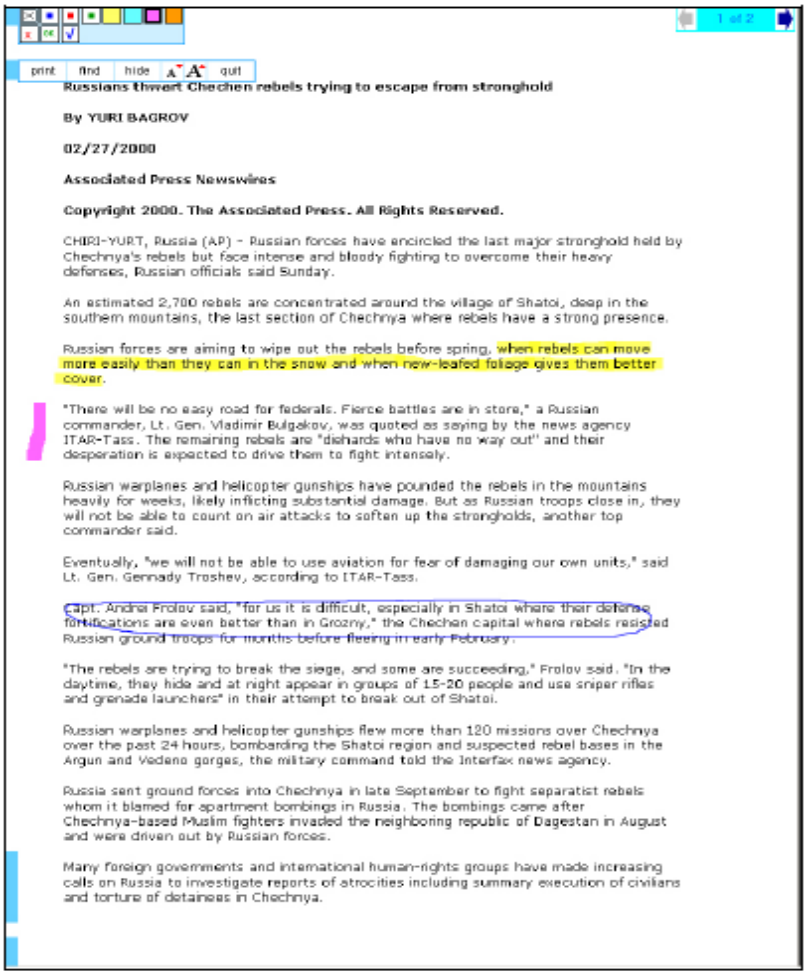


Figure 2.11: XLibris built in 2002

- TAS simply anchors annotations to associated selected text (Brush et al., 2001). That is, a portion of text is highlighted, and this highlight is then attached to the associated portion of text.

- Callisto generally anchors user annotations to neighboring text; for example, the anchor of a circled passage is the words surrounded by the circle, the anchor of an underline or highlight is the words intersected by the annotation, and a marginal note or symbol is associated with lines of text overlapped by the vertical position of the annotation (Bargeron & Moscovich, 2003).

- Microsoft Office products use a similar approach to Callisto to identify the anchor of every annotation.

- IAWD implements an ink-to-document association algorithm in Javascript (Flanagan, 2001) using DOM and DHTML to associate underlying HTML elements with ink (Ramachandran & Kashi, 2003). For example, a circled passage is associated with the text ranges within the boundary box of the annotation which are queried using the DOM APIs (Ramachandran & Kashi, 2003).

- ScreenCrayons allows users to annotate whatever interests them and then use that information to summarize what was captured (Olsen et al., 2004). In other words, users capture interested sections as an image then annotate in the image. ScreenCrayons is to extract natural boundaries from the image that can then be associated with annotations (Olsen et al., 2004).

- RCA determines the anchor of a comment by its linker, which associates a comment with a specific code line (Priest & Plimmer, 2006). A linker can be a line or a circle. If its linker is a line, then the comment is associated with the code line closest to the start point of the linker, otherwise, the comment is associated with the line closest to the mid-point of the circle's height (Priest & Plimmer, 2006).

Reflowing annotations

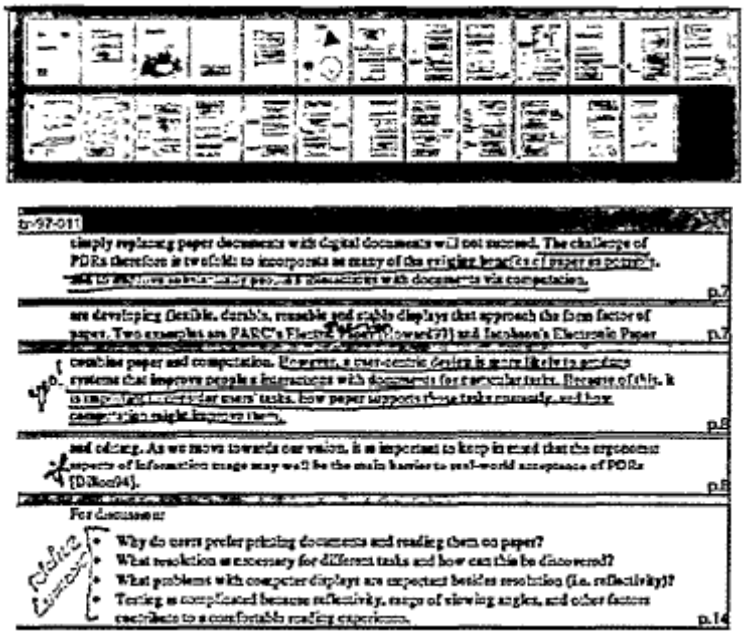
Reflowing annotations aims to maintain the consistency of annotations with their underlying content when the annotated document is modified. Reflowing annotations is performed after grouping and anchoring annotations.

Digital annotations can be shared among users. Different users may have different devices with different screen sizes, resolutions, and fonts, possibly influencing the display of the annotated document. To keep the consistency of annotation and its attached content, annotation tools must enable annotations to be repositioned to match changes in the layout or font of the document. Two tools with this function exist. XLibris (Golovchinsky & Denoue, 2002) and Callisto both reposition annotations according to their anchor information, with the difference between them being that Callisto must group and classify annotations before defining the anchor of an annotation, while XLibris is free of this requirement.

Some users want to edit and modify the document content. It is difficult to preserve the relationship between the annotation and its underlying content. However, Microsoft Office products and RCA use anchor information to maintain the relationship between annotations and associated context. Microsoft Office products move annotations up or down vertically according to information provided by the anchor. However, Microsoft Office products can not delete annotations when their anchors are deleted. RCA groups comments based on their spatial and temporal properties, and then attaches each group to a linker (Priest & Plimmer, 2006). The anchor of every group is decided by the linker, and existing annotations can then be moved up or down vertically based on anchor information. RCA also deletes annotations whose underlying code is deleted or modified.

Supporting navigation system

Navigation system can help users review and locate existing annotations. XLibris (Schilit et al., 1998) provides a document view that shows thumbnails of all pages (Figure 2.12a), and a Reader's Notebook that extracts annotated clippings and lays them end-to-end (Figure 2.12b). This approach is only suitable for textual documents with pagination boundaries. TAS (Brush et al., 2001) provides a list of anchors of all annotations for a web page and the contents of note annotations. This approach is only suitable for lists annotations in a single file.



a

b

Figure 2.12: XLibris's navigation system. (a) the document view shows page thumbnails, including annotations, for the entire document; (b) the Reader's Notebook shows annotated clippings of documents laid end-to-end.

However, a program document is normally very long and split among many files. Navigation support of a code annotation tool thus should provide users with an outline of existing annotations and information on how many annotations exist and where, as well as which code file is annotated. In addition, such navigation support should assist users in easily locating a specific annotation. No currently available annotation tools provide such navigation support.

Sum up

From the above detailed comparison among annotation tools, seven technical challenges in developing a code digital ink annotation tool were explored.

- Seamlessly integrating the tool inside an IDE to allow users to annotate directly over the code window. Then users enjoy both IDE and annotation support.
- Enabling annotated documents to display without losing any information and to allow users to edit annotated documents while reviewing or annotating them.
- Enabling users to make any kinds of annotations at anywhere on the digital documents.
- Grouping individual ink strokes into annotations.
- Anchoring the grouped annotations to their desired context.

- Reflowing annotations to maintain their consistency with their anchored context and their visual meanings and styles.
- Providing a navigation system to help users review and locate annotations easily and conveniently.

2.4 Summary

This chapter described related work on traditional ink annotation, digital ink annotation and digital ink annotation tools. The literature on traditional ink annotation shows that readers enjoy making various kinds of annotations throughout paper documents, different kinds of annotations have different functions, and annotations containing words provide better understanding and communication and collaboration when annotated documents are shared. The literature on digital ink annotation demonstrates that readers are gradually becoming accustomed to reviewing and annotating digital documents on computers along with the development of technology in hardware and software. Finally, digital ink annotation tools were described and the technical challenges and valuable ideas in developing a program code digital ink annotation tool were explored. The next chapter presents the proposed design of a digital annotation tool for program code documents inside an IDE.

Chapter 3

Proposed Design

The above discussion of traditional ink annotation and digital ink annotation tools reviewed existing annotation techniques and tools and identified technical challenges and requirements that must be met by a successful code digital ink annotation tool. This chapter describes the requirements of a program code digital ink annotation tool that meets these challenges and requirements. Crucially, this tool must integrate seamlessly into an IDE to allow users not only to make annotations on code but also to implement all functions provided by the IDE such as programming, debugging, executing etc.

A program code digital ink annotation tool must support four main functions: annotations must be free-form and modifiable, annotations must reposition automatically in response to changes in the underlying text, handwritten comments must be able to be recognized to provide a clear view, and navigation support must be provided to help users review, select and find annotations.

3.1 Extending the IDE

IDEs do not support digital ink annotation. The following three methods thus are used to review program code documents.

- First, programmers read and annotate source code using pen and paper. However, this approach suffers a critical drawback resulting from the unique feature of source code: it is non-linear in that it is arranged in logical classes and methods (Priest & Plimmer, 2006). Source code cannot be simply read sequentially like a book because users must search through numerous pages to identify classes or methods. Furthermore, printed-out source code is static and does not permit programmers to debug or execute code.

- Second, programmers employ a separate application to make annotations on code when reading, such as Penmarked (Plimmer & Mason, 2006). The weakness of this approach

is that it only deals with static documents and opens files as a text file, so the code (Java, C# etc.) can not be compiled or run under this application. Users therefore have to jump between the IDE and this application to run and annotate code.

- Finally, an annotation tool (such as RCA (Priest & Plimmer, 2006)) that works inside an IDE allows users to review and annotate simultaneously within the IDE. However, although this approach permits users to review and annotate code in the IDE, currently available tools have failed to integrate seamlessly into IDEs. For example, in RCA, users can make annotations in the ink window rather than directly in the code window, and must switch between the ink and code windows to annotate and run or modify code.

To preserve the functionality of IDEs and provide digital ink annotation support, the third approach, in which a program code digital ink annotation tool resides directly within an IDE, is one of the best approaches for simultaneously running, modifying, reviewing, and annotating source code. Employing a program code digital ink annotation tool as a plug-in or extension of an IDE permits users to review and annotate code inside the IDE and to simultaneously implement all IDE's functionality (such as debugging, editing, and execution). This approach requires the IDE to be extensible to enable developers to extend its functionality and incorporate other functions. Currently two of the most popular IDEs are Visual Studio (Microsoft Visual Studio, 2005) and Eclipse (Clayberg & Rubel, 2006). Visual Studio supports numerous programming languages such as VB, C++, C#, J# and so on. Eclipse is specific to Java language. This project selected Eclipse as the IDE within which the program code digital ink annotation was integrated. Eclipse was selected because it is an open-source software, thus allowing users access the underlying framework and to easily customize and extend the existing framework.

To allow the IDE to perform normally while supporting digital ink annotation, a program code annotation tool must integrate into the IDE seamlessly. Such seamless integration allows users to annotate directly over the code window, and to review, annotate, and edit via the same window -- the code window.

Since the annotation tool is integrated into an IDE, this tool can support documents that are supported by the IDE. For example, Eclipse supports Java language, and so too does the

annotation tool. Additionally, both the annotation and IDE support enable users to edit documents during annotation.

3.2 Basic functionality

With pen and paper users can make any type of annotations anywhere within a paper. To be successful a program code digital ink annotation must have the same ability. Digital documents have potential advantages over paper documents in that modifying such documents does not deface their appearance. Moreover, digital ink annotations can be stored for later review.

3.2.1 Free-form and modifiable annotations

Free-form annotations comprise words, symbols and text selection marks, and can be made anywhere without restrictions of shape or content (Schilit et al., 1998; Golovchinsky & Denoue, 2002). Schilit et al. (1998) argued that unstructured and idiosyncratic marks have rich meanings to the annotators and thus support episodic memory, although they may be meaningless to a computer. Also annotators tend to categorize annotations by color, type, or some other approaches (Marshall, 1997; Heinrich & Lawn 2004). To meet this requirement, a program code digital annotation tool must provide various colored pens and highlighters to increase the flexibility of expression. Furthermore, such a tool must enable users to annotate wherever they wish. This requirement is achieved by overlaying the code window with a transparent canvas that holds ink annotations and associated annotations to the underlying code by making the code window and the transparent canvas scroll together. This approach allows users to annotate anywhere inside the code window; there is no restriction of the pagination boundary in the code window as there is in document formats such as PDF that set artificial restraints on user positioning of the annotations.

When annotating on paper documents with pens, it is hard to modify existing annotations. Furthermore, it is almost impossible to delete ink annotations without leaving unpleasant traces. Modifying ink annotations usually involves concealing or crossing out the original annotations and then rewriting them, yielding a messy result (Priest & Plimmer, 2006). A program code digital ink annotation tool eliminates these difficulties and increases the flexibility of modifying annotations. This tool enables users to efficiently and cleanly erase and modify existing

annotations, and also provides users with a means of readily moving, recoloring, and resizing annotations.

3.2.2 Save and load

The ability to save and load digital ink annotations is one of key abilities that a digital ink annotation for program code must support. When a user first selects a program code file to annotate, a new annotation file is created and stored as part of the development project. Once the user finishes the annotation, the annotation file must be saved for later review. When a user wants to review or annotate a program code file that has already been annotated by that user or others, the existing annotation file is loaded to allow the user to review or continue annotating the code file.

3.3 Reflowing annotations

One of the main features of a program code digital ink annotation tool is the ability to deal with dynamic digital documents. Since annotation is usually a function of some type of review it must be expected that text inside the documents will be changed. This requires existing annotations reflow to remain consistent with the underlying code when the code file is modified. “Reflow” in digital ink annotation refers to the repositioning or deletion of existing annotations in response to changes in the dynamic underlying context (Priest & Plimmer, 2006). Reflowing annotations requires grouping ink strokes. Annotations belonging to the same underlying context must be grouped together by considering location and temporal properties of the ink strokes that comprise the annotation (Golovchinsky & Denoue, 2002; Bargeron & Moscovich, 2003). When the underlying code moves up/down, any attached annotation groups must also move. Meanwhile, when the underlying code is deleted, any attached annotation must be either deleted or archived.

3.3.1 Grouping annotations

Grouping individual ink strokes when they are made is the first step in successfully reflowing annotations. Each group of ink strokes is then associated with the same underlying context.

Temporal order and spatial arrangement must be considered when grouping ink strokes (Chiu & Wilcox, 1998; Golovchinsky & Denoue, 2002; Bargeron & Moscovich, 2003).

The temporal property of ink strokes refers to the time period between an ink stroke being made and the next ink stroke occurring. If this time period is within 500ms, the two strokes are assumed to be grouped together based on the findings of Golovchinsky and Denoue (2002) that strokes belonging to the same word are made within approximately 500 milliseconds. However, the temporal approach is not an appropriate method of grouping ink strokes in certain situations, for example, when users dot an i after completing a word. In these cases, spatial proximity may be a better criterion for grouping (Bargeron & Moscovich, 2003). Spatial proximity refers to grouping an ink stroke being made to an existing group that is spatially adjacent to the stroke, namely, considering spatially adjacent annotations to belong to the same group. The spatially adjacent group is that closest to the ink stroke.

3.3.2 Anchoring grouped annotations

The second step in successfully reflowing annotations is to attach each group to its intended context such that the context can be recovered even the document layout, format, or content change (Bargeron & Moscovich, 2003). From the perspective of annotators, every mark is associated with a particular piece of content, such as a line of text, a paragraph, a set of words, images etc, which becomes anchor of the annotation (Golovchinsky & Denoue, 2002). Program code files have three notable features: words do not wrap around to new lines; each code statement occupies one line; and each program code file contains extensive repeated code (e.g. loop, return or catch statements) (Priest & Plimmer, 2006). Considering these three features, an annotation is anchored to a specific piece of code as follows:

- A highlight or underline is anchored to a code line that is intersected by that highlight or underline.
- A straight line is attached to a code line that is closest to the start point of the straight line.
- Other annotations are associated with a code line that is closest to the middle of its bounding box.

3.3.3 Repositioning grouped annotations

The final step is to reposition the annotations while the annotated document is modified. A program code file is arranged line by line. Individual code line cannot be rearranged as sentences can, and typically lines are not wrapped when resizing the code window. Therefore, it is unnecessary to consider the issue of reflowing annotations horizontally. Instead it is only necessary to focus on moving annotations up or down inside the code window (Priest & Plimmer, 2006).

Users may modify text documents to delete certain words or paragraphs, reword certain sections, move sections of texts or paragraphs, or add more words or paragraphs. Programmers modify program code files for the same reasons. Four possibilities must be considered in relation to the modification of program code files. First, when code lines with attached annotations are deleted, the annotations must also be erased. Additionally, other annotations located below these deleted code lines must be moved up vertically. Second, each code line exhibits a strict structure of methods and variable names (Priest & Plimmer, 2006). When a code line is replaced, its attached annotations thus must be deleted because the new line probably presents different functionality, resulting in the loss of the context and meaning of the annotation (Priest & Plimmer, 2006). Third, when a line or a group of lines is moved to a new location, its attached annotations must be repositioned accordingly. Finally, when a line or a group of lines is added, all annotations that are below the added lines must move down.

3.4 Recognizing handwritten annotations

Most research on digital ink recognition has focused on handwriting recognition (Plamondon & Srihari, 2000). In the case of reviewing program code, most annotations take the form of notes, meaning there are numerous handwritten comments. Handwriting characteristics typically differ enormously between different individuals. Some may be legible, while others may be illegible. Also since handwritten comments can only be stored as images, they cannot be sorted, searched, and so on. To improve the legibility of handwritten comments and extend the ability to sort and search handwritten comments, automatic handwriting recognition is a key function of a program code digital ink annotation tool.

3.5 Navigation system

Previous techniques used for building navigation support are not suitable for code. Program code documents have no concept of pages or pagination boundaries. Moreover, program code files are structured by classes and procedures. Therefore, it is not feasible to break code into pages for editing or review. Furthermore, a programming project generally contains numerous program code files. The navigation must support all code files in the project to allow users to readily and easily navigate between them. IDEs frequently incorporate a navigation system to assist programmers in learning which code files exist in the project and how classes are related in a code file, and to locate specific classes.

This thesis contends that it may be useful to provide a similar annotation navigation system to users. A program code file often extends over many lines and cannot be fully displayed on a screen. Therefore it is necessary to scroll the screen upwards/downwards to review existing annotations, which is very inconvenient. Users can employ the navigation system to easily locate a specific annotation inside the code window. Furthermore, the navigation system assists users in determining which code files have been annotated, as well as what annotations have been made and where.

3.6 Summary

This chapter presented an overview of the requirements of a program code digital ink annotation tool for use in an IDE. A digital ink annotation tool for program code is presented that supports free-form and modifiable annotations, reflowing annotations, recognition handwritten annotations, and a navigation system. This proposed annotation tool provides users with an environment for annotating naturally and intuitively inside an IDE. The next chapter details the use of these requirements to develop a program code digital ink annotation tool called CodeAnnotator within Eclipse.

Chapter 4

Implementation

The previous chapter detailed the core requirements for designing a program code digital ink annotation tool within an IDE. Based on these requirements, we designed and developed an annotation tool called CodeAnnotator within Eclipse. This chapter first introduces CodeAnnotator. CodeAnnotator is then described in detail, followed by the approaches attempted to annotate on program code files within Eclipse. Finally, the functionality of CodeAnnotator is detailed, including editing free-form annotations, preserving annotations, grouping annotations, anchoring annotations, reflowing annotations, handwriting recognition, and navigation system.

4.1 Introduction

CodeAnnotator is developed within Eclipse as a set of plug-ins for two reasons. First and foremost, Eclipse is a popular open source IDE and together with its extensible architecture, this allows application developers to extend Eclipse and build tools that integrate seamlessly with the Eclipse environment (Aeppli, 2005). Second, Eclipse supports Java, which is widely used in teaching departments and industry. CodeAnnotator allows users to make annotations directly over Java files inside Eclipse.

CodeAnnotator is implemented in Java using the Java API, Eclipse platform, and a set of plug-ins including the Plug-in Development Environment (PDE), the Graphical Editing Framework (GEF) and Draw2D. The Eclipse platform gives access to the navigation and outline systems, information related to opened windows (including window name and type, content, and so on) and event notifications (such as file content change events, file open and save events, window creation and activation events etc.). PDE provides a set of tools for creating, developing, testing, debugging, and deploying Eclipse plug-ins (Melhem & Glozic, 2003). The basic structure of CodeAnnotator is implemented using GEF because it supports developers in creating a rich graphical editor for visually creating and editing models (Zoio, 2004). For example, when an

object changes state, GEF becomes aware of the change, and performs appropriate actions, such as redrawing a figure in response to a move request (Aniszczyk, 2005). GEF ships with a painting and layout plug-in called Draw2D that provides figures and layout managers which form the graphical layer of a GEF application (Lee, 2003). GEF and Draw2D allow for manipulating annotations such as adding, moving, modifying, recoloring, erasing, selecting, grouping and repositioning. These services enable the effective and efficient development and implementation of CodeAnnotator.

CodeAnnotator can be used on either Tablet PCs or desktop computers fitted with a tablet USB input device. However, Tablet PCs are the better choice for three reasons. First, Tablet PCs offer a pen and paper like appearance, which allowing users to read and annotate naturally and intuitively. Second, unlike desktop computers, with Tablet PCs users can directly annotate onto the output screen with a digital pen. Third, Tablet PCs are more convenient to travel with and annotation is often undertaken when away from a desk.

4.2 Overview

Figure 4.1 shows the final version of CodeAnnotator which provides an annotation window to hold digital ink annotations in Eclipse. When the Eclipse with the CodeAnnotator plug-in is opened, the CodeAnnotator navigator can be displayed. When a user loads a program code file of a project that has never previously been annotated and selects an annotation mode by clicking the "  " icon ("Pen" button) that has been added to the Eclipse toolbar, a new annotation window is created based on the current code window in Eclipse and an outline window is simultaneously created. The annotation window is then saved as an annotation file with the same name as the corresponding program code file, differing only in the file extension. For example, if the program code file is called "Test.java", then its corresponding annotation file is called "Test.ink". This annotation file is then stored in the project directory. Otherwise, if the program code file already has a corresponding annotation file, then that annotation file is loaded.

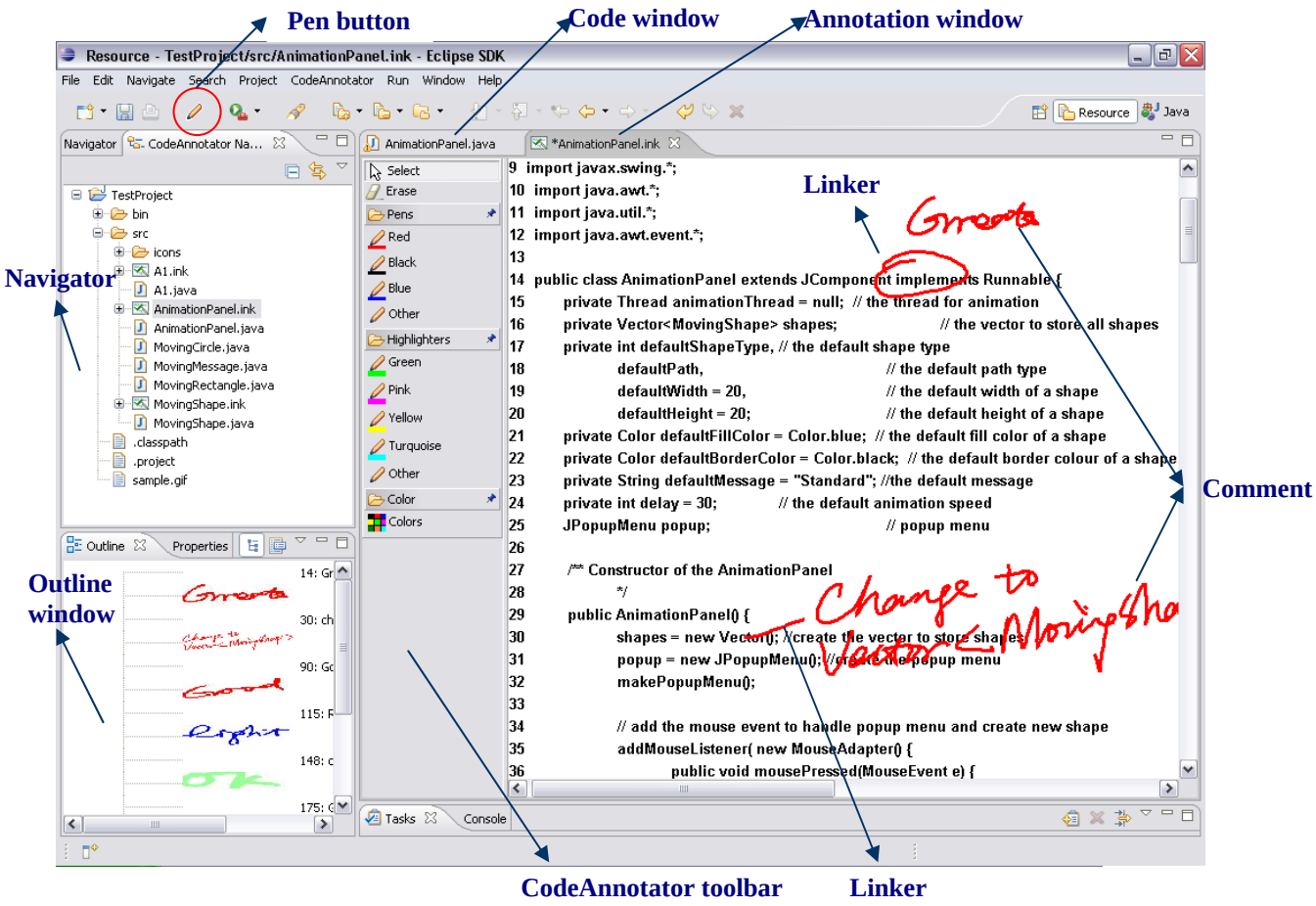


Figure 4.1: CodeAnnotator user interface

Before continuing to describe CodeAnnotator, it is important to explain what components an annotation has and the meanings of the terminology adopted in this thesis. Figure 4.2 shows an example of an annotation. An *annotation* includes a *linker*, a *comment* formed by *strokes*, and a specific piece of code.

A *stroke* is the ink mark left by a pen between it touching the screen and being lifted off the screen. Each stroke is represented as a series of x,y points that are joined by straight lines. The frequency of the points means that the lines appear as natural hand-drawing with smooth curves.

A *linker* is a stroke made to link a comment to a specific piece of code. The linker can be any kind of symbol such as a straight line, circle, square, bracket and so on. Strokes made after the linker must be grouped.

A *comment* is a group of strokes associated with a specific piece of code.

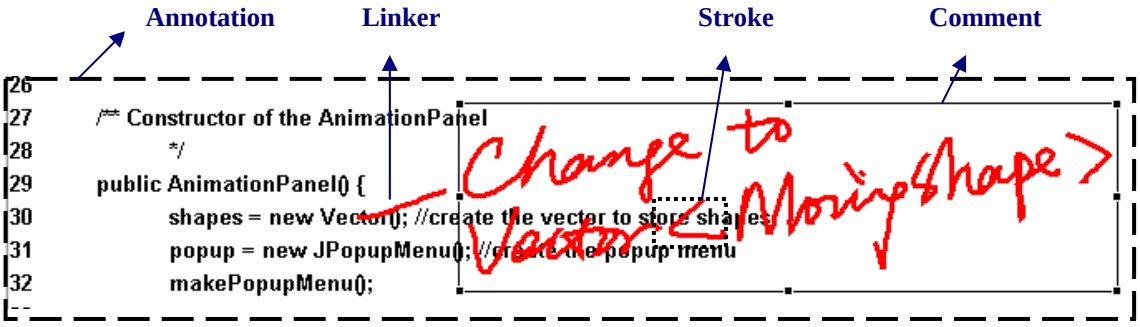


Figure 4.2: Example of an annotation

Users wanting to make annotations on the code must go to the corresponding annotation window. A linker must be created before writing any comments regarding a specific piece of code. Strokes made after the linker are grouped according to their spatial and temporal properties (Chiu & Wilcox, 1998; Golovchinsky & Denoue, 2002; Bargeron & Moscovich, 2003). Finally, the piece of code to which the comment is attached is decided by the location and properties of the linker (as detailed in the "Reflowing annotations" section). When users make an annotation, the information related to that annotation (such as the code line number of the annotation attached, the comment etc.) is displayed in the outline window. The annotation window with annotations and the outline window with the graphical list of existing annotations are saved in the annotation file. These annotations can also be edited, modified and repositioned (as discussed in the following sections). Specific tools are required for editing and modifying annotations.

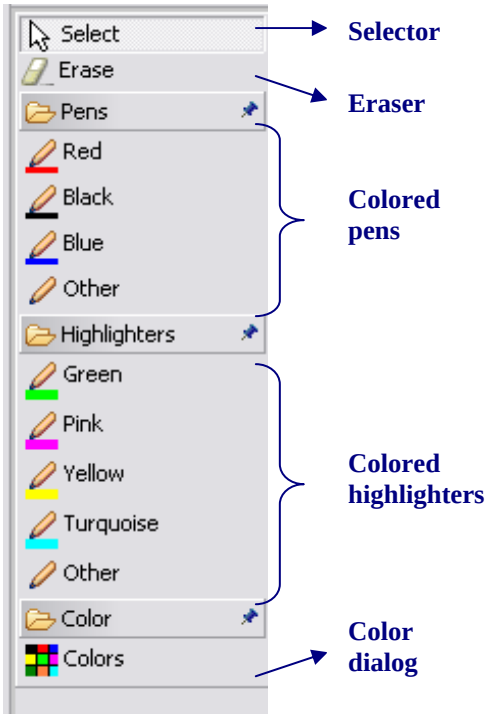


Figure 4.3: CodeAnnotator's annotation tools

CodeAnnotator provides tools to help users edit and modify annotations. Figure 4.3 above shows the annotations tools. The tools are located in the Palette of the left side of the annotation window and include a selector, an eraser, colored pens, colored highlighters, and a color dialog. These tools represent four kinds of annotation modes: selection, deletion, editing, and recoloring. Users can select a linker or comment by the selector. When the selector button is clicked, the annotation window operates in the selection mode. In this mode users cannot add, delete or recolor annotations, but can resize and relocate the selected linker or comment. The eraser enables users to delete unwanted annotations. When the eraser is selected, users enter the deletion mode. In this mode users can make different colored and styled annotations by selecting colored pens or highlighters. When a pen or highlighter is chosen, the annotation window operates under the editing mode, in which users can add a new annotation. Finally, clicking the color dialog initiates the recoloring mode, in which users can recolor an existing annotation by choosing a color from the color dialog.

4.3 Eclipse extension

The goal of a program code digital ink annotation tool within an IDE is to allow users to make annotations directly on the code in the IDE. In other words, users can view or change code and

make annotations in the same window. This requires building a transparent layer to hold digital ink annotations. This transparent layer must seamlessly and completely overlay the code window, and the annotation layer and code window must scroll up and down together. This research experimented with three approaches for supporting fully integrated digital ink annotation over the code window.

The first approach was to build a transparent annotation window that completely overlays the code window and the two windows can scroll together. The idea of this approach was to create a transparent annotation window and allow the annotation and code windows to act as a single window. In this approach it was possible to review, edit and annotate the code directly on the transparent annotation window. With this approach we thought it would be possible to make a window transparent because some methods (such as OS.SetWindowLong and OS.SetLayeredWindowAttributes etc.) offered by the Microsoft Windows Operating System can make Windows transparent. However, these Microsoft Windows OS methods created problems by making the entire Eclipse and everything within it transparent. Attempts were thus made to set the Alpha value of figures to make them transparent. However, setting Alpha value to zero can make fingers inside a window transparent but cannot make the window itself transparent. Moreover, it was found that it remained impossible to review, edit and annotate code on the same window, and instead it was necessary to return to the code window for editing. Because in Eclipse other opened windows are hidden when a window is activated and, only the activated window can be edited. Additionally, it is complex to simultaneously move the scrollbar of the two windows. Consequently, this approach was abandoned.

The second approach attempted in this research was to create an annotation window that totally mimicked the code window and was editable. The expectation with this approach was that the annotation window contained a copy of code from the code window, this copy shared the same layout and appearance as the code in the code window, and users could change code directly on the annotation window and the code window would automatically update the changes to maintain the consistency with the annotation window. Copying the code into the annotation window is simple. However, maintaining the layout and appearance of the program code is difficult because each file type is displayed via specific layout and appearance strategies in Eclipse. For example, a Java file has its own layout and font and color rules for displaying the code in Eclipse, and these rules cannot be preserved when copying the code to other windows. Consequently, it was necessary to produce rules specifically for displaying the copied code in

the annotation window. Extending the annotation window from the code window, the annotation window can only deal with text and not with figures or drawing as the code window is actually a text editor in Eclipse. Extending the annotation window from GEF, it is complicated to program rules as GEF treats everything as figures rather than text. Furthermore, synchronizing the code and annotation windows was challenging, although it is easy to make the annotation window editable. Owing to the problems above this approach was also abandoned.

The final approach attempted in this research was implemented in CodeAnnotator. We took the idea from the second approach of building a separate annotation window, but the copy of the program code was held as the window background. With this approach the code in then annotation window is not editable, and users have to change code in the code window. It is not technically difficult to maintain consistency between the annotation and code windows and the annotation window is updated automatically. The annotation window is extended from GEF, allowing figures or drawings to be dealt with effectively and efficiently. While this approach is a compromise, users can review and annotate code on the annotation window. However, this approach has two drawbacks. First, the code displayed in the annotation window loses its font and color rules, being shown in plain text in the annotation window. This drawback can be reduced in the future by programming the display rules for the code in the annotation window. Second, the code and annotation windows are not physically related. Thus the code in the annotation window must be refreshed every time that in the code window changes.

4.4 Anchoring annotations

Isolating an annotation from its intended context will confuse reviewers and hinder their understanding of its real meaning. For example, an annotator place a “?” beside a word as a reminder to check the meaning of that word, but if the content of the document is invisible, reviewers will only see an isolated question mark and thus will not understand the meaning. Therefore, an annotation becomes meaningful only when attached to specific context. To maintain its relationship with its associated context, the annotation must be reflowed when the document changes layout or undergoes editing. This requires anchoring the annotation to its intended context (for example, the location to which it belongs in the document, the content with which it is associated and so on) such that the context can be recovered even if the document layout, format, or content changes (Bargeron & Moscovich, 2003).

Program code documents differ from textual format documents (e.g. Word, Text). Program code files are structured line by line; a “hard return” is inserted at the end of each line, and restricts word wrapping between lines (textual format documents usually perform word wrapping). Therefore, there is no need to worry too much about changes in the program code file layout or format. Each code line in the code document has its own distinct meaning. Normally, programmers make comments to specific lines or sections of code. These comments must be attached to specific code lines to maintain consistency with the underlying code when the line of code moves up or down. Linkers are used to anchor each annotation to its intended code line. A linker is a bridge between a comment and its intended code, namely an ink stroke is used to link a comment to a specific code line (Priest & Plimmer, 2006). The linker is made before the comment. In other words, before writing a comment, users must make a linker first.

CodeAnnotator implements the RCA approach (Priest & Plimmer, 2006) to anchoring annotations by using linkers. RCA only provides two types of linkers, line linkersand circle linkers. Moreover, RCA requires users to make linkers in the “linker mode” and comments in the “inking mode”. This confuses users with regard to which mode they are in. This approach is extended to allow linkers to be any kinds of symbols and to allow users to make linkers or comments under one mode, the editing mode. Linkers can be a straight line, circle, square, bracket, or any other symbols. Each linker is treated differently and thus it is necessary to identify linkers before proceeding. A feature of a straight line is that its start and end points touch the left and right border of the bounding box (See Figure 4.4). A circle or a square can be differentiated from a line by measuring whether the distance between the first and last points (ab) is less than 25% of the hypotenuse of the bounding box (a’b’) (See Figure 4.5). Otherwise if ab is greater than a’b’ but less than the hypotenuse of the bounding box, then the linker must be considered a bracket or others (See Figure 4.6).

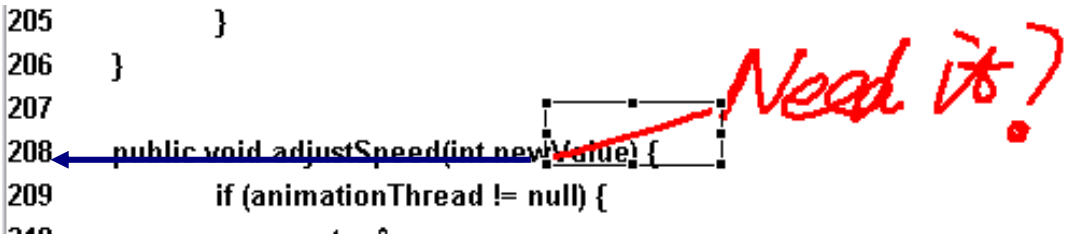


Figure 4.4: A straight line linker

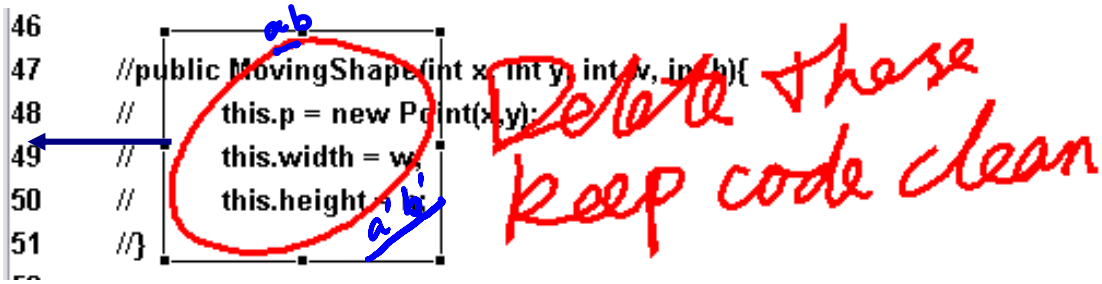


Figure 4.5: A circle linker

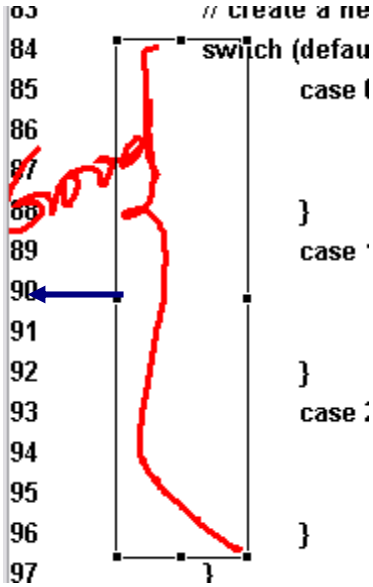


Figure 4.6: A bracket linker

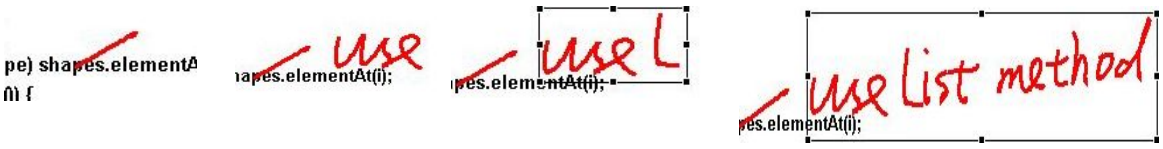
If the linker is a line, then its corresponding annotation group is linked to the code line closest to the start point of the linker. For example, Figure 4.4 shows the line linker connects the comment to code line 208. For other linkers the corresponding annotation group is attached to the line closest to the middle point of the bounding box. For instance, Figure 4.5 displays that the circle linker links the comment to code line 49, and the bracket linker in Figure 4.6 associates the comment with code line 90. An annotation group is a comment made after the linker and is a group of stokes. Successfully reflowing annotations requires grouping strokes made after a linker.

4.5 Grouping annotations

A comment generally comprises of several words and symbols. Meanwhile, a word comprises several letters. Whether or not a word can be completed in a single stroke largely depends on

user writing behavior and word formation (for example, in a word containing a t or i, users must go back to cross the t or dot the i). It is impracticable to predict how a user will write a word since everyone has unique and variable writing behavior. If a word is completed by three strokes, it is necessary to let the computer realize that the three strokes form the word and that the word comprises the three strokes as the computer is unaware of this. In other words, while a user writes or draws on a document, the annotation tool must be able to group individual ink strokes into comments, at roughly the same level of abstraction and with the same accuracy as a human would (Bargeron & Moscovich, 2003).

Ink strokes are grouped based on their temporal and spatial properties (Chiu & Wilcox, 1998; Golovchinsky & Denoue, 2002; Bargeron & Moscovich, 2003). Temporal grouping is based on the time between the current and last ink strokes. Golovchinsky and Denoue (2002) found that the average time between strokes that belong to a single comment is approximately 500 milliseconds. For example, in the case of a line linker and the comment “use List method”, after making the line linker, the user writes “use” with a single stroke in less than half a second, then “use” belongs to the linker. The “L” is made soon afterwards, and then the “L” and “use” are grouped together and the group is assigned to the linker, with “ist” being made shortly after that, and also assigned to the group (See Figure 4.7). However, if “L” is written more than half a second after “use”, if the user makes another linker within 500 milliseconds after completing the comment, or if the user goes back to dot the i and cross the t after completing the comment, then the spatial property of the stroke is used to either assign it to an existing group or classify it as a new linker.

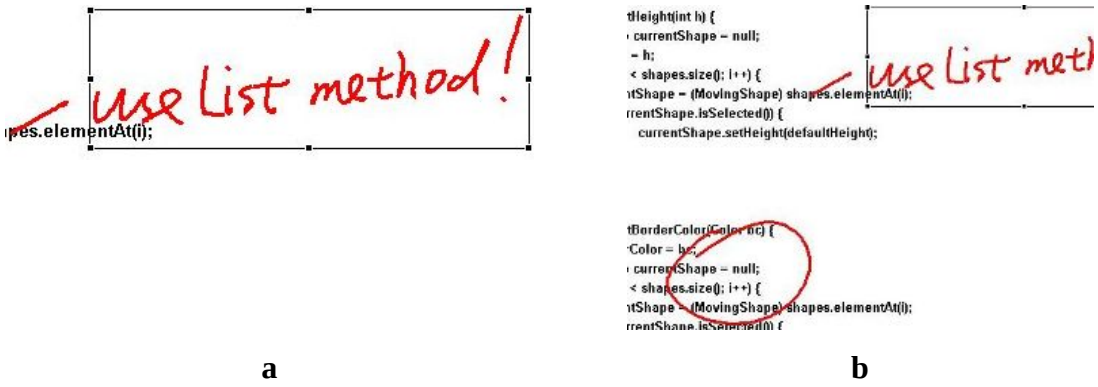


a **b** **c** **d**

Figure 4.7: Temporal grouping strokes. (a) a line linker is made; (b) “use” is made; (c) “L” is made; (d) the comment “use List method” is completed

Spatial grouping considers the distance between the current and the previous ink strokes. Spatially adjacent stokes are considered to belong to the same group. For example, if an annotation including a line linker and the comment “use List method” already exists, the user makes a new ink stroke, and if this stroke is distant from the existing comment, then it is

considered to be a new linker, but if it is close to the existing comment, then it is considered part of the existing comment. (See Figure 4.8)



a **b**

Figure 4.8: Spatial grouping strokes. (a) "!" is made near the comment; (b) a circle is made far from the comment

CodeAnnotator uses the above two approaches to decide the group to which a new stroke belongs or whether it is a new linker. For the temporal approach, following the creation of a linker, if the subsequent stroke is created in less than half a second and is spatially adjacent to the linker, then it belongs to the linker. Moreover, if the subsequent stroke is created in less than half a second and is spatially adjacent to the previous stroke, then both this stroke and the previous stroke are grouped together and this group belongs to the linker. If a new stroke is added within half a second and is near the previous stroke, then it is part of the group. However, if a new stroke is added after half a second or is far from the group but being made within half a second, then we employ spatial approach to tell which group the new stroke belongs to or whether it is a new linker.

For the spatially grouping approach, the position of the new stroke (StrokeA) is calculated; at the coordinate (x4, y4), it is assumed to have height h4 and width w4. According to this calculated position, the closest annotation group is identified and the stroke is then assigned to that group. For example, given the situation where three groups already exist: the first group (Group1) is located at the coordinate (x1, y1), and as size h1, w1, the second group (Group2) is located at the coordinate (x2, y2), and has size h2, w2, and the third group (Group3) is located at the coordinate (x3, y3), and has size h3, w3. The following five situations thus must be considered.

1. If StrokeA overlaps one of the three groups horizontally and vertically, then StrokeA belongs to that group. For example, Figure 4.9 shows that StrokeA overlaps Group1 horizontally and vertically, StrokeA is part of Group1.

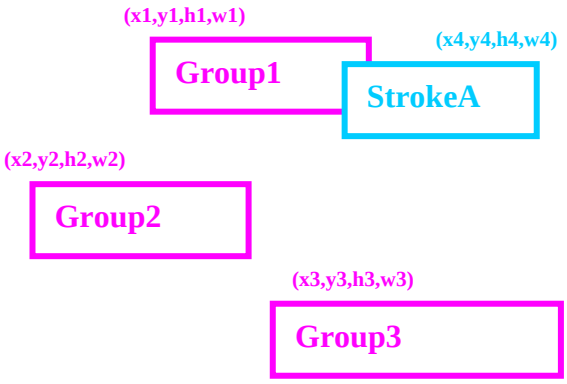


Figure 4.9: Stroke overlaps a group horizontally and vertically

2. If StrokeA partially overlaps one of the three groups vertically and the distance between StrokeA and the group is less than 100 pixels, then StrokeA belongs to that group. This distance is limited to 100 pixels owing to annotators normally insert words inside the comment or add words near the comment. For example, Figure 4.10 shows that the range from y4 to y4+h4 only partially overlaps the range from y2 to y2+h2 and the distance D4-2 is less than 100 pixels, and thus StrokeA belongs to Group2.

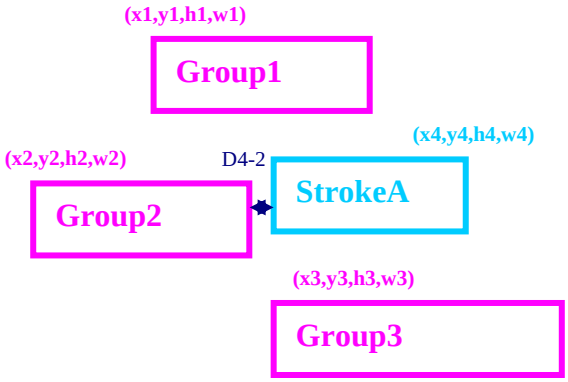


Figure 4.10: Stroke overlaps a group vertically

3. If StrokeA overlaps multiple groups vertically, then the distances between StrokeA to each group are checked to determine which is closest:

    a) If StrokeA follows or precedes the overlapped groups, StrokeA belongs to the closest group and the distance between the two is less than 100 pixels. For example, Figure 4.11 shows that x4 is bigger than x2 and x3, and that if x4 is

closer to x2+w2 than to x3+w3, then StrokeA belongs to Group2, otherwise it belongs to Group3, namely, D4-3 is less than D4-2 and is less than 100 pixels, so StrokeA is part of Group3. Figure 4.12 shows that D4-2 is less than D4-3 and also less than 100 pixels, so StrokeA belongs to Group2.

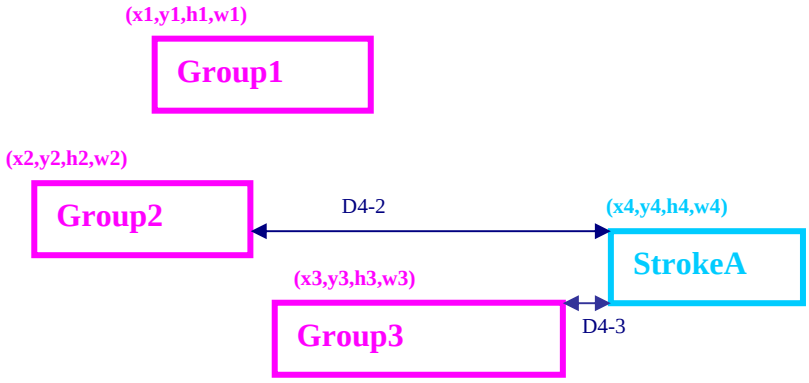


Figure 4.11: Stroke overlapping groups vertically and after the overlapped groups

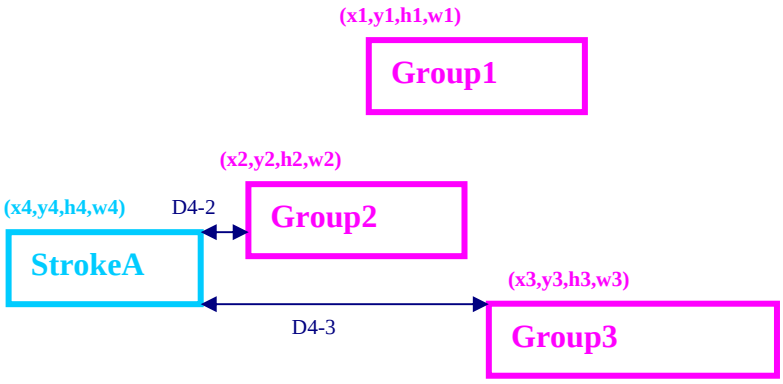


Figure 4.12: Stroke overlapping groups vertically and before the overlapped groups

b) If StrokeA follows an overlapped group and precedes another overlapped group, then StrokeA belongs to the group before StrokeA if the distance between StrokeA and the group is less than 100 pixels. Grouping StrokeA into the group before it due to annotators are more likely to append comments to the existing annotation closely. For example, Figure 4.13 shows that x4 is larger than x2 and smaller than x3 and D4-2 is less than 100 pixels, and thus StrokeA belongs to Group2.

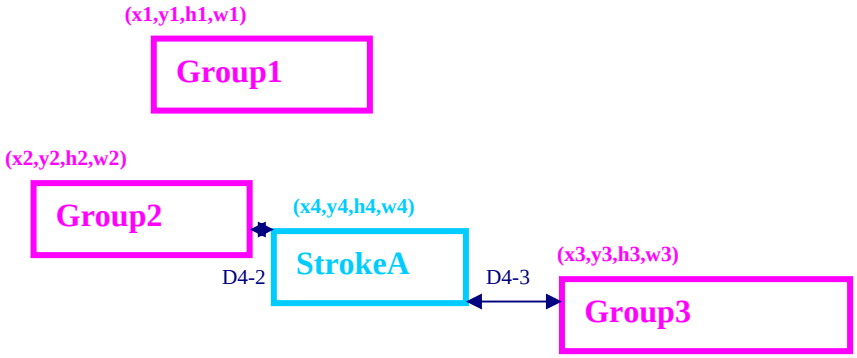


Figure 4.13: Stroke overlapping groups vertically and before one and after another

4. If StrokeA partially overlaps groups horizontally, then the distances between StrokeA and each group are checked, and StrokeA is considered to belong to the closest group if the distance between them is less than 10 pixels as words are normally added beside the comment. For example, Figure 4.14 shows that the range from x4 to x4+w4 partially overlaps the ranges from x1 to x1+w1, x2 to x2+w2, and x3 to x3+w3, we can see StrokeA is closest to Group2 as D4-2 is less than D4-3 and D4-1 and is less than 10 pixels, so StrokeA is part of Group2.

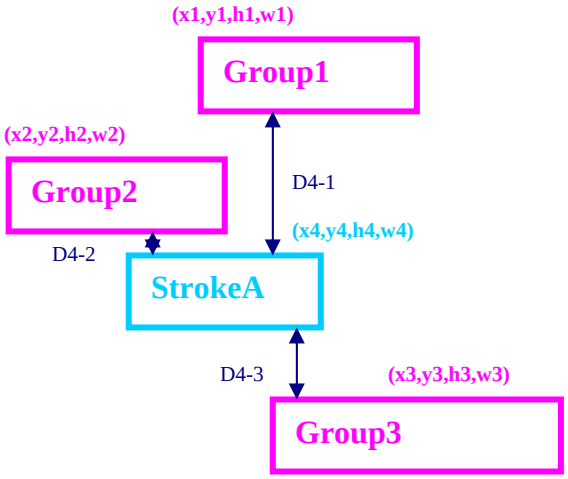


Figure 4.14: Stroke overlapping groups horizontally

5. Otherwise, StrokeA is considered to be a new linker. For example, Figure 4.15 shows that StrokeA is a new linker although it overlaps several existing groups horizontally because D4-2 exceeds 10 pixels.

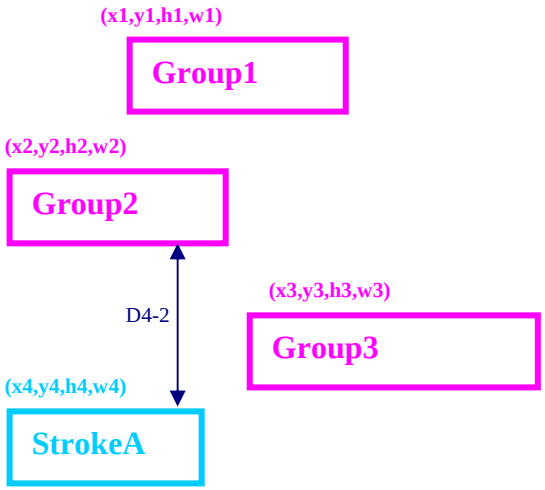


Figure 4.15: Stroke overlapping several groups horizontally but the distance exceeds 10 pixels

The original CodeAnnotator only considered the above five situations. However, the usability study (that is discussed in Chapter 5) reflected that certain strokes were incorrectly grouped. To increase the accuracy in grouping stokes, this research extends the fourth situation into two sub-situations. The fourth situation thus is as follows:

4. If StrokeA partially overlaps groups horizontally and does not overlap other groups either horizontally or vertically, the distance between StrokeA and each overlapped group is calculated to get the smallest distance (M1). The distance between StrokeA and each unoverlaped group is then calculated to get the smallest distance (M2). Comparing M1 and M2 yields the minimum distance. The group to which StrokeA belongs and whether it is a new linker is then determined based on the following two conditions:

    a) If M1 is less than M2 and M1 is less than 35 pixels, StrokeA belongs to the overlapped group closest to StrokeA. For example, Figure 4.16 shows that StrokeA overlaps Group1 and Group2, and that D4-2 is less than both D4-1 and D4-3, and is also less than 35 pixels, so StrokeA belongs to Group2.

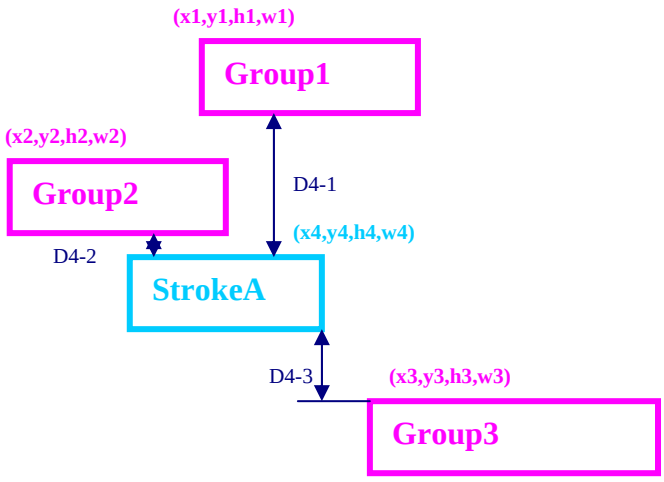


Figure 4.16: Stroke overlapping groups horizontally and the distance is less than 35 pixels

b) If M2 is less than M1 and M2 is less than 35 pixels, StrokeA belongs to the unoverlapped group closest to StrokeA if StrokeA is ahead or after the group within a distance of 100 pixels. For example, Figure 4.17 shows that StrokeA overlaps Group2, but D4-2 is larger than D4-3, which is less than 35 pixels, so StrokeA belongs to Group3 if H4-3 is less than 100 pixels.

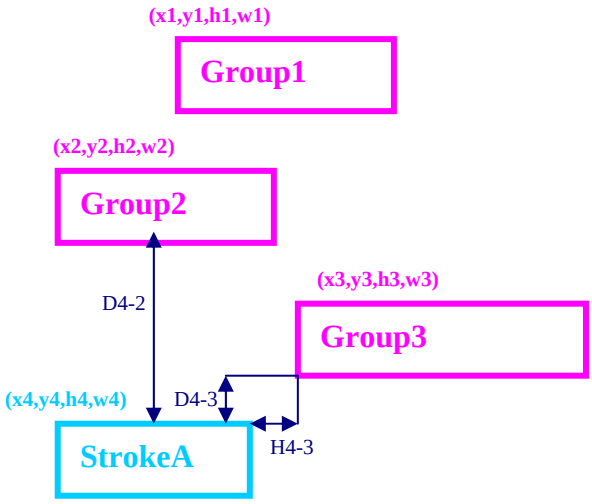


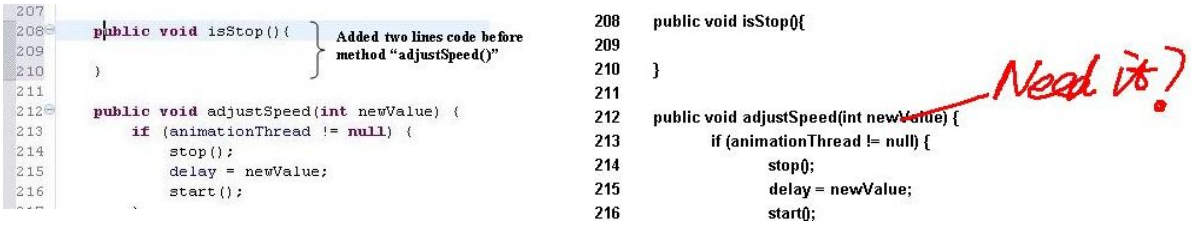
Figure 4.17: Stroke overlapping groups horizontally but belonging to an unoverlapped group

Otherwise, if a new stroke cannot meet the above three situations and the revised fourth situation, then that stroke is a new linker.

4.6 Reflowing groups

In the pen and paper approach, printed documents are fixed and not editable. All ink annotations are affixed to a specific location on a paper. This approach works adequately for users reviewing textual format documents since such documents are normally read like a book and do not require producing a result like program code documents. However, it is difficult for users to generate the result of the program code when reading the program code document as print-outs. The result of a program code document is changed if a single code line is changed. Comments on the program code documents are usually intended help users fix errors in the program code. Users then change the code according to the comments. The comments must remain associated with the relevant code after the code is changed. The most notable advantage of the program code digital ink annotation tool is allowing users to edit the underlying document while reviewing or making annotations. Reflowing existing annotations is necessary to maintain consistency with the underlying code when it is edited.

After anchoring linkers to specific code lines and grouping strokes after linkers, it is possible to reflow annotations. Only the vertical reflow needs to be considered since code is line-based. Existing annotations are reflowed according to their anchor information, the attached code and its line number. For example an annotation including a line linker and the comment “Need it?” is made and anchored to code line 208 (See Figure 4.4), while simultaneously two code lines are added before line 208 in the code window. On returning to the annotation window it is then found that the annotation moves down and remains attached to the same code line. (See Figure 4.18)



a **b**

Figure 4.18: Reflowing annotations. (a) add more code lines; (b) annotations are reflowed

Users do not need to save changes they make to the code in the code window. CodeAnnotator automatically saves changes to the code when users switch to the annotation window. Whenever

the annotation window is active, the tool checks whether the code in the code window and that in the annotation window are the same. If differences exist, the annotation window deletes the original code background and replaces it with a copy of the current code from the code window. All existing annotations are deleted or repositioned based on the attached code line and the code line number. Three situations need to be handled (Priest & Plimmer, 2006). First, when several code lines are added ahead of a code line with annotations, this code line and its associated annotation group move down together. Second, when several code lines are deleted ahead of an annotated code line, this code line and its corresponding annotation group move up together. Third, when a annotated code line is deleted, its associated annotations are also deleted.

4.7 Basic functionality

The two essential advantages of digital ink annotations are: annotations can be easily modified and edited, and annotations can be preserved for later review. These two advantages are discussed in detail below.

4.7.1 Editing free-form annotations

The ability to edit annotations is a very important function in annotation tools. Particularly, users may wish to reposition comments. CodeAnnotator supports moving specific comments by selecting and dragging, with the linker moving together with the comment. Users also can select the linker and drag it to a new position, in which case its related comment simultaneously moves to that new position. The linker position is recalculated after the linker and comment are relocated.

An annotation can be deleted or erased by selecting it with the eraser tool. Users can also use the undo/redo function to undo or redo the most recent activity. Thus if users delete or move an annotation, that annotation can be restored to its original position by clicking the undo button. If the redo button is clicked, the annotation is deleted again or moved to the previously selected new location. If users add a new stroke, the new stoke can be deleted by clicking the undo button and then restored by clicking the redo button.

Annotations can be written in different colors by selecting a color before writing. There are no restrictions on the kinds of strokes user can make and where they can be made, though strokes must be in the annotation window. Moreover, users can recolor an existing annotation by first selecting the annotation and then selecting a different color from the color dialog.

4.7.2 Preserving annotations

The traditional pen and paper approach has difficulty in preserving annotations for long periods. An annotation tool must be able to preserve digital ink annotations for sharing or future viewing. CodeAnnotator has the ability to maintain consistency between the code when the annotation window is closed and that when loading its corresponding annotation file because it is possible for annotators or reviewers to change the program code file after closing the annotation window.

Save

A code annotation must save digital ink annotations since annotations are normally reviewed by multiple individuals. For instance, in the education area, teachers or markers can annotate student program code files and then return the annotated files to students. CodeAnnotator saves digital ink annotations in an annotation file. The annotation file and its corresponding code file share the same name but different extension types. The extension for the annotation file is “ink”. The information saved includes annotations, their locations, and the code each annotation attached to. Digital ink annotations can be saved via two methods. First, when the annotation window is open, users can save annotations by clicking the save button in the Eclipse toolbar. Second, when the annotation window is closed, all annotations are saved automatically.

Load

There are two situations in which the annotation file can be loaded. First, if no corresponding annotation file exists for the program code file users want to annotate, they must open the code file in Eclipse, create a new annotation file and display that file as an annotation window in Eclipse by clicking the “” icon (“Pen” button) in the Eclipse toolbar or the submenu “CodeAnnotator” under the “CodeAnnotator” menu in Eclipse (See Figure 4.4). The annotation

file is saved in the same directory as the code file, and is displayed in the CodeAnnotator Navigator (detailed in the "Navigation system" section). Second, if the program code file has a corresponding annotation file, this file can be loaded by simply double-clicking the annotation file in the CodeAnnotator Navigator. CodeAnnotator then simultaneously loads the program code and annotation files. Alternatively, users can open the program code file first and then click the "Pen" button or "CodeAnnotator" submenu, which will cause CodeAnnotator to automatically search for the corresponding annotation file and load it in Eclipse. Users can check whether the program code file has a corresponding annotation file in the CodeAnnotator Navigator (See Figure 4.19).

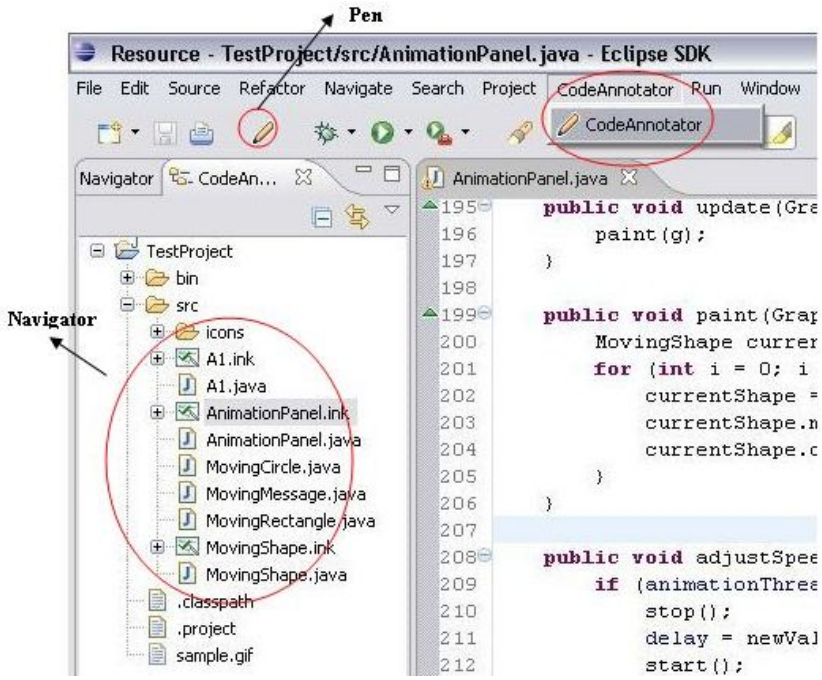


Figure 4.19: Ways of loading annotation files

Maintain consistency

An annotation becomes meaningful through association with a specific piece of code. If the piece of code is moved while the annotation remains in the same place, then the annotation loses its meaning. This phenomenon can occur in numerous situations, such as when code in the program code file is changed after the closure of its corresponding annotation window, when code is sent to others for feedback and so on. All of these situations may create differences between the code from the program code file and that in the annotation window. One way to eliminate this problem is to lock the program code file. However, this approach is impractical

for programmers. CodeAnnotator can completely resolve this issue and maintain the functionality of Eclipse (such as editing, debugging, executing etc.). CodeAnnotator allows the annotation file to keep a copy of the annotated code. When the annotation file is loaded, CodeAnnotator then checks whether the code of the corresponding program code file differs from the saved code. If they are different, the tool repositions all existing annotations to maintain consistency with the attached piece of code. The reflowing issue was discussed in the 4.6 section.

4.8 Handwriting recognition

Handwriting recognition functions present a challenge because they require separating writing and iconic annotation. None of digital ink annotation tools described in section 2.3 offer digital ink recognition functionality. Recognizing handwriting may help users in reviewing handwriting comments, anchoring comments to suitable content, and sorting or searching comments.

CodeAnnotator employs the handwriting recognition system provided by the Microsoft Vista Operating system using a Java interface developed by Coyette et al. (2007). After finishing an annotation in the editing mode, the user changes to the selection mode by clicking the “Selector” button on the annotation toolbar. The user points the cursor over the annotation, and the recognized handwriting text appears below the annotation and is displayed in the Outline window. For example, when clicking the “Selector” and pointing the cursor over the comment “Good to have it”, then the recognized text “Good to have it” appears below the comment and is also shown in the Outline window, which also displays recognized text for other annotations. (See Figure 4.20)

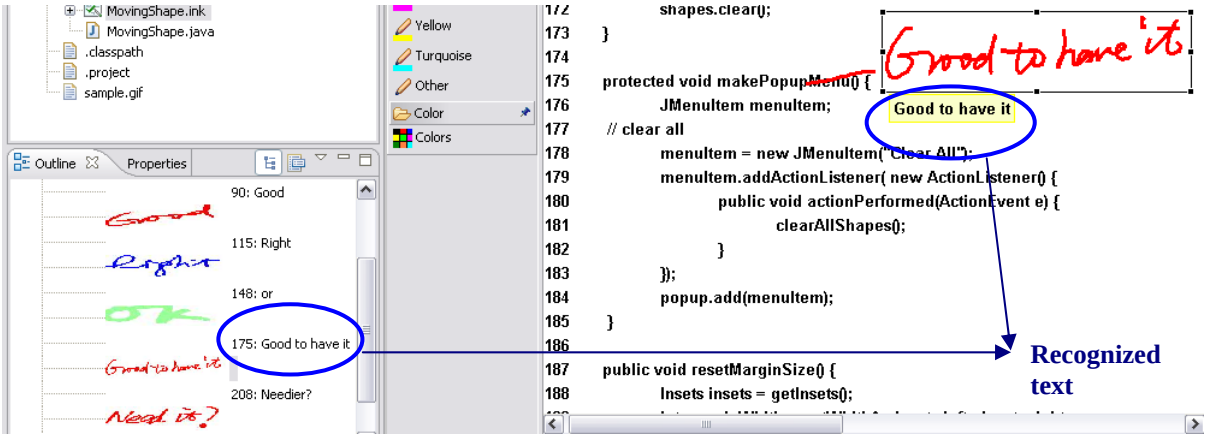


Figure 4.20: Handwriting recognition

Originally, CodeAnnotator performed handwriting recognition whenever a stroke was made, and the recognized handwriting text was displayed in the graphical list of annotations in the Outline window. However, the usability study (discussed in Chapter 5) reflected that this approach negatively impacted smooth and clear writing performance. To overcome this deficiency, handwriting recognition is conducted only when the user places the mouse over the annotation under the selection mode and saves the annotation window. When the user saves the annotation window, all existing annotations are recognized and the recognized handwriting texts are displayed in the Outline window.

4.9 Navigation system

Content of textual format documents is generally displayed in a page by page manner. Users thus can locate content using page numbers. However, code documents have no pagination boundaries. A code file is formed from classes and methods and is normally too long to be displayed on a single screen. Additionally, a code project typically contains more than one program code file. Most IDEs provide a navigation system to help users to find program code files and locate pieces of code within them. Annotations made on a program code file share the same features as the program code file. Moreover, code projects generally involve annotations to multiple program code files. To assist users in finding annotation files and readily and easily locating annotations in those files, w a navigation system is provided to fulfill a similar function to IDEs.

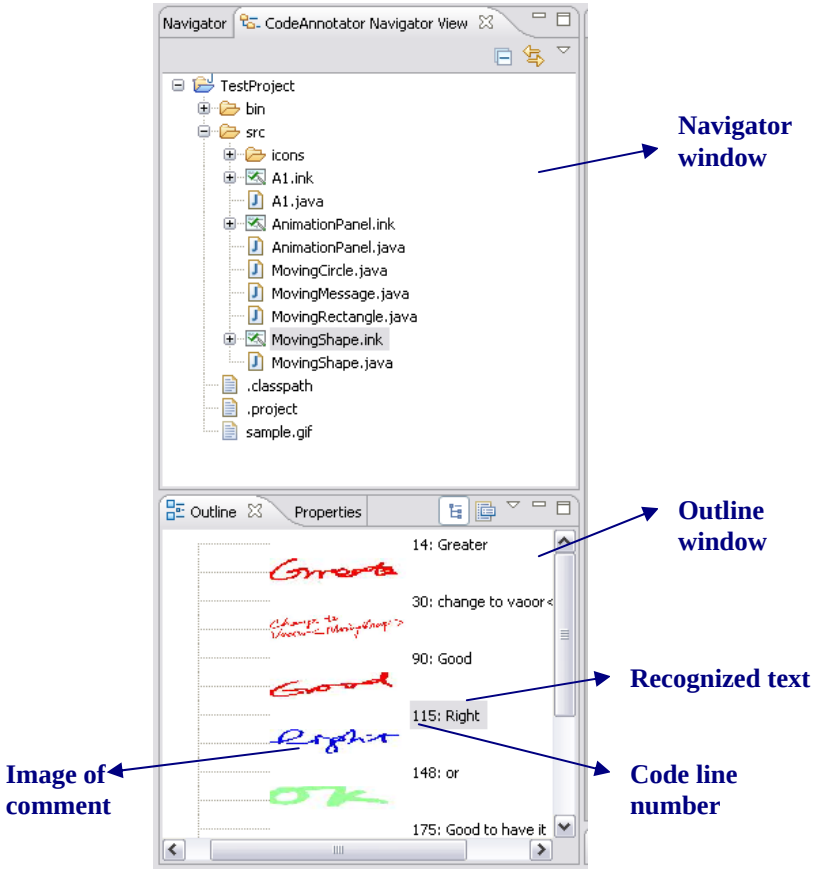


Figure 4.21: CodeAnnotator navigation system

The navigation system in CodeAnnotator includes a Navigator window and an Outline window (See Figure 4.21). The Navigator window tells users which program code files have been annotated and provides a simple graphic list of annotations for every annotation file. Meanwhile, the Outline window guides users in locating the position of a specific annotation inside the annotation window, and provides a detailed graphical outline of existing annotations. The graphic lists presented in the two windows contain the line number of the code associated with each annotation, the text of each recognized handwriting comment, and the image of each comment (i.e. a graphical copy of the comment). (See Figure 4.21) For example, clicking the "Plus" symbol near AnimationPanel.ink brings up a simple graphic list of the annotations it contains, and if AnimationPanel.ink is opened from the Navigator window by double-clicking then, AnimationPanel.java is opened together with its Outline window and a detailed graphic list of annotations is displayed. (See Figure 4.22)

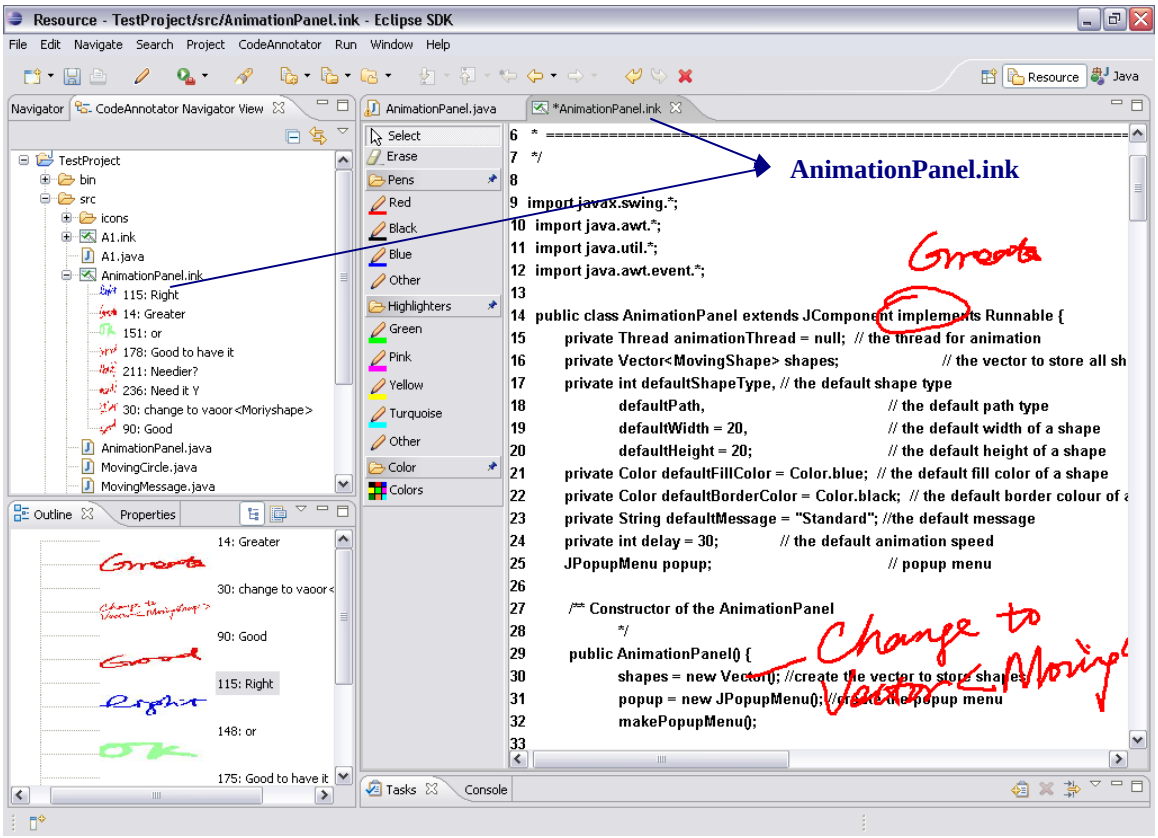


Figure 4.22: Navigator and Outline windows of AnimationPanel.ink

Users can decide whether or not to open an annotation file by reviewing the annotations contained in that file in the Navigator window. If an annotation file is found worthy of detailed review, users can directly open that file in the Navigator window. The corresponding program code file is automatically opened simultaneously with the annotation file, and an Outline window is also automatically opened and shows a graphical outline of existing annotations in this annotation file. When users make an annotation in the annotation window, a copy of this annotation and its corresponding information including the associated code line number, recognized handwritten comments in text form and an image of the original handwritten comments are displayed in the Outline window. When users select a specific annotation inside the Outline window, the annotation window scrolls to the page containing that annotation. For example, if the user selects “Right” in the Outline window, the annotation window scrolls down to display that annotation. (See Figure 4.23)

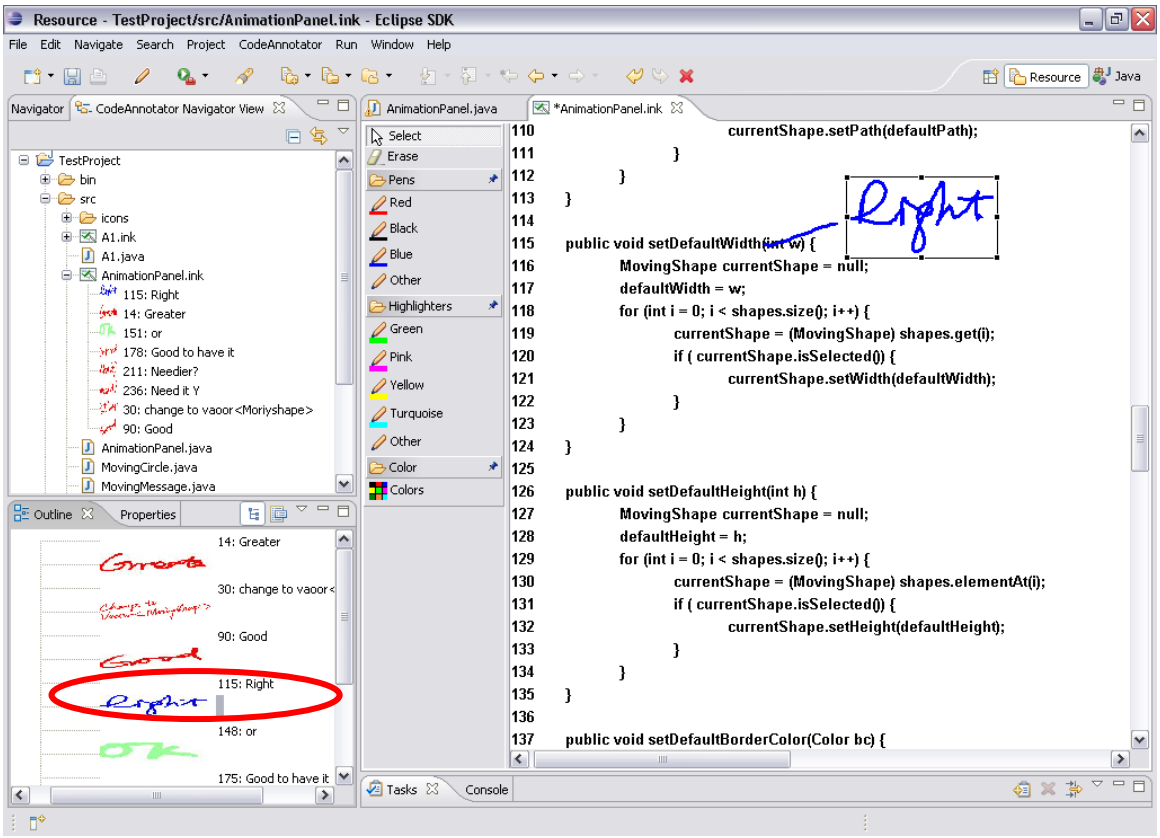


Figure 4.23: Select an annotation from the Outline window and display it in annotation window

The Outline window also contains another approach to help users scroll the annotation window to the intended location. This approach is called “Eagle Eyes”. After clicking the “” icon in the Outline window, the Outline window shows an encapsulated annotation window. Moving the mask up/down or left/right causes the annotation window to display the area covered by the mask. For example, if the user moves the mask to the area containing the annotation “Right”, the annotation window scrolls down to display this area. (See Figure 4.24)

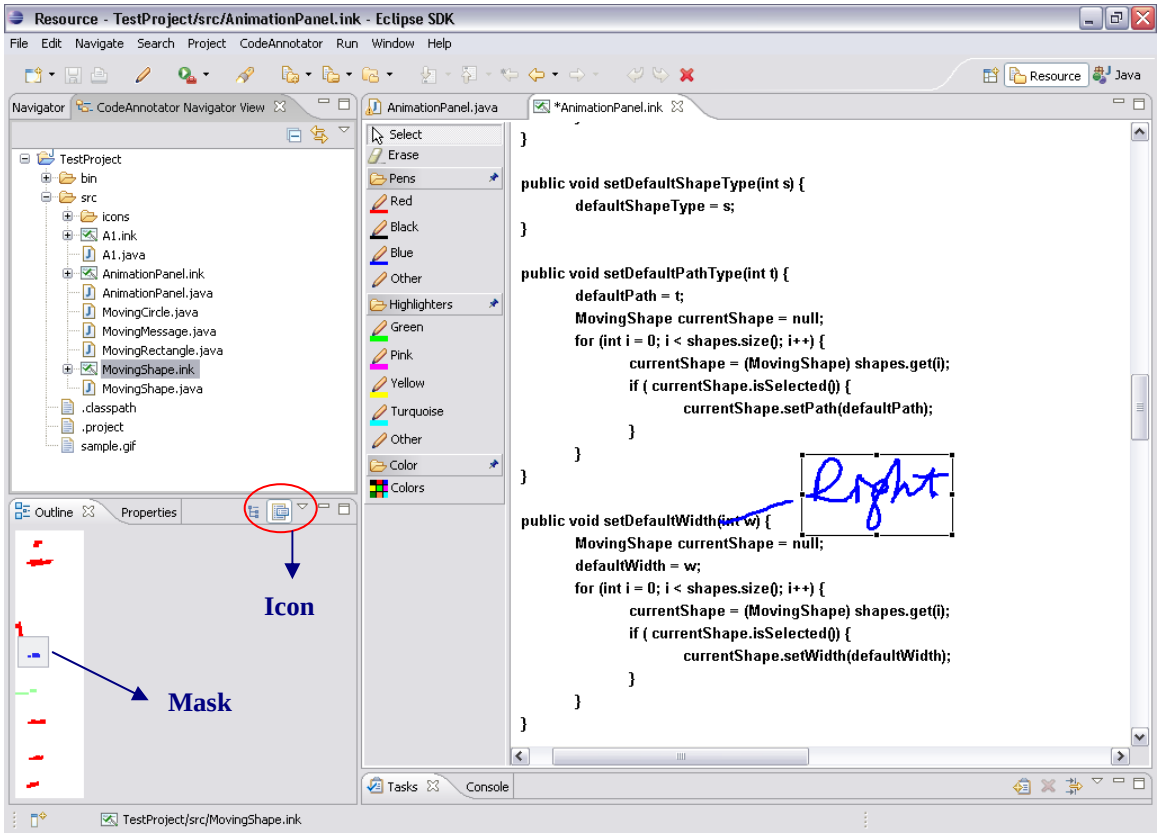


Figure 4.24: Eagle Eyes

4.10 Summary

This chapter detailed the functionality of CodeAnnotator. CodeAnnotator is integrated into the Eclipse environment so that programmers continue to enjoy normal IDE support while annotating or reviewing annotations. CodeAnnotator allows users to review and make free-form annotations in the annotation window, edit and modify existing annotations, edit code in the code window, and reflow existing annotations when the underlying code is edited. Additionally, CodeAnnotator provides the recognition of handwritten comments and a navigation system to help users conveniently review and search annotations. CodeAnnotator is ready to use in its current form. However, before releasing it to the public, it must be evaluated to identify its strengths and limitations.

Chapter 5

Evaluation

CodeAnnotator is designed and developed to provide users with an electronic environment for annotating Java documents within Eclipse. A usability study was carried out to measure the effectiveness of the implementation of CodeAnnotator and how well various functions of CodeAnnotator work together. This chapter starts with the design of the usability study and then findings of that usability study.

5.1 Usability study

Usability study is a technique used in design to help developers identify product weaknesses and improve them by measuring product usability or user-friendliness (Dumas & Redish, 1999). A usability study was conducted to understand real user reactions to the functionality of CodeAnnotator. The study included creating free-form annotations, editing annotations, grouping strokes, reflowing annotations, and a navigation system. The usability study aimed to assess the strengths and limitations of CodeAnnotator, and focused on answering the following questions:

- Are users able to find pens/highlighters with different colors?
- Are users able to make annotations anywhere inside the annotation window?
- Are users able to make free-form annotations?
- Do users have difficulty changing existing annotations (i.e. recoloring, deleting, and moving)?
- Are the strokes grouped correctly?
- Can CodeAnnotator reflow existing annotations correctly when the underlying code is edited?
- Does the navigation system assist or hinder users in seeking and reviewing annotations?

To answer these questions the usability study was designed as detailed in the following section. Appendix 1 presents a full copy of the usability study plan is in.

5.1.1 Usability study design

Study participants were selected from the following groups: students, markers and teachers. All participants were required to have experience with programming Java and using an IDE such as Visual Studio or Eclipse. Participants were acquired via email or invitation notices or by word of mouth. Six participants plus a pilot tester participated in this study, including two females and five males. One participant was a teacher, one was a marker, and the others were students. All participants had at least one year of experience in programming Java and using IDEs. Only one participant had rarely used an IDE for programming Java. However, no participants had previously used a digital pen for marking/making comments on digital files. Most participants had only sometimes used computer tools for making comments on digital files. Furthermore, three participants had never used a pen input on a computer and the others had only limited experience of such input.

This study was conducted in a room equipped with a tablet computer and an observer chair. The keyboard of the computer was covered, but a wireless keyboard and mouse were available. The tablet computer was set up with Eclipse, CodeAnnotator, and Morae Recorder. Morae Recorder simultaneously recorded participant voices, screen text, keystrokes, and mouse clicks. Each participant took around 30 minutes to complete the study.

In the study, participants were asked to play two roles – a marker and a student. First, participants played the role of a marker and had to complete four tasks. Participants were asked to make five annotations on the Java document that simulated a student assignment. After making the five annotations, they then had to delete, recolor, and move an existing annotation. Finally, they were required to append words to an existing comment. Second, participants played the role of a student and were required to complete a task involving changing program code based on marker comments. Specifically, participants were required to locate two annotations from the existing annotation files and edit the corresponding Java files according to those annotations. (The five study tasks are detailed in Appendix 1.) The five tasks had to be completed in 11 minutes, with this time limit being established from the pilot study.

Before carrying out these tasks, participants were asked to read the participant information sheet and sign the ethical approval form. (They are in Appendix 1.) Participants were then asked to fill out a short pre-study questionnaire (See Table 5.1) gathering background information.

1. What is your gender?
2. What is your current university status, a student or marker or teacher?
3. How much experience do you have in programming Java?
4. How much experience do you have in IDE (e.g. Visual Studio, Eclipse)?
5. How often do you use an IDE for programming Java?
6. How often do you use a digital pen for marking/making comments on digital files?
7. How often do you use computer tools for making comments on digital files?
8. How often do you use a pen input on a computer?

Table 5.1: Pre-study questionnaire

After completing the pre-study questionnaire, participants were trained to know what CodeAnnotator does, to familiar with the equipment, and to know how to make annotations, edit existing annotations, group and reflow existing annotations, and use the navigation system to review and locate a specific annotation. (The training script is in Appendix 1.) Once participants felt comfortable with the interface and functionality of CodeAnnotator, they were presented with a task script detailing the five tasks to be accomplished. Next, participants attempted these tasks. While completing these tasks, participants were observed and their emotions and mannerisms noted, as well as their voices, mouse movements and keystrokes, and the screen all were recorded for later analysis.

After concluding the five tasks, participants were asked to complete a post-study questionnaire (Table 5.2) comprising eight questions to gather their responses regarding the tasks and the overall performance of CodeAnnotator. Participants were asked to rate the eight questions and responded on a five point scale where a rating of “1” indicated “strongly disagree”, “2” was “disagree”, “3” was “neutral”, “4” was “agree”, and “5” indicated “strongly agree”. In other words, “5” was the highest rating, “1” was the lowest rating. The questionnaire also asked participants to provide comments or recommendations regarding CodeAnnotator.

Q1: It was easy to make annotations.
Q2: It was easy to change existing comments (recolor, delete and move).
Q3: The navigation system aided me seeking and reviewing annotations.
Q4: This tool is helpful and useful for me to make comments on code files.
Q5: This exercise was enjoyable.
Q6: I thought that this tool was easy to use.
Q7: I found the various functions in this tool were well integrated.
Q8: I would like to use this method of interaction in the future.

Table 5.2: Post-study questionnaire

Before any participants carried out the study, the pilot tester conducted a pilot testing to test the suitability of the usability questions (for example, questions and tasks may be too difficult or easy or long or short) and to make changes as appropriate. After all participants completed the study, the accuracy rates of grouping strokes, anchoring groups and reflowing groups were analyzed to evaluate the functionality of grouping, anchoring and reflowing annotations. Furthermore, to evaluate other aspects of the functionality of CodeAnnotator, participant questionnaire feedback was analyzed, as well as participant comments, emotions and mannerisms recorded during the study.

5.1.2 Usability study results

CodeAnnotator provides a successful tool for making digital ink annotations over electronic code documents within an IDE. Users can write annotations while reviewing, editing or debugging code. The usability study found that CodeAnnotator was easy to use and participants would like to use it in the future. Overall participant feedback was positive. The study also solved some of the weaknesses of the tool, and identified interesting ideas for its future improvement.

One of participants commented that “The application was great! It would be nice to use in the future”. Mean scores were calculated for each question based on participant ratings from the post-study questionnaire and the following figure was thus obtained (Table 5.3 and Figure 5.1). Figure 5.1 shows that Q2 has the highest score (4.5), indicating participants largely agreed that it was easy to recolor, delete and move existing annotations. Meanwhile, Q5, Q6, and Q7 all

score 4.17, suggesting that participants agreed that this study was enjoyable, the tool was easy to use, and functions in the tool were well integrated. The remaining questions have scores of around 3.5, indicating that participants were inclined to agree that it was easy to make annotations (Q1), the navigation support aided them seeking and reviewing annotations (Q3), this tool was helpful and useful for them to make annotations on code files (Q4), and they would like to use this tool in the future (Q8). The results are detailed from five aspects: integrated into Eclipse, making free-form annotations, editing annotations, navigation system, and grouping, anchoring and reflowing annotations.

	Participant1	Participant2	Participant3	Participant4	Participant5	Participant6	Average
Q1	3	5	2	5	4	3	3.67
Q2	3	5	5	5	5	4	4.50
Q3	5	3	3	5	5	2	3.83
Q4	4	5	3	3	4	2	3.50
Q5	4	5	3	4	5	4	4.17
Q6	4	5	4	4	5	3	4.17
Q7	4	5	3	5	4	4	4.17
Q8	4	5	2	3	4	4	3.67

Table 5.3: Ratings of each question

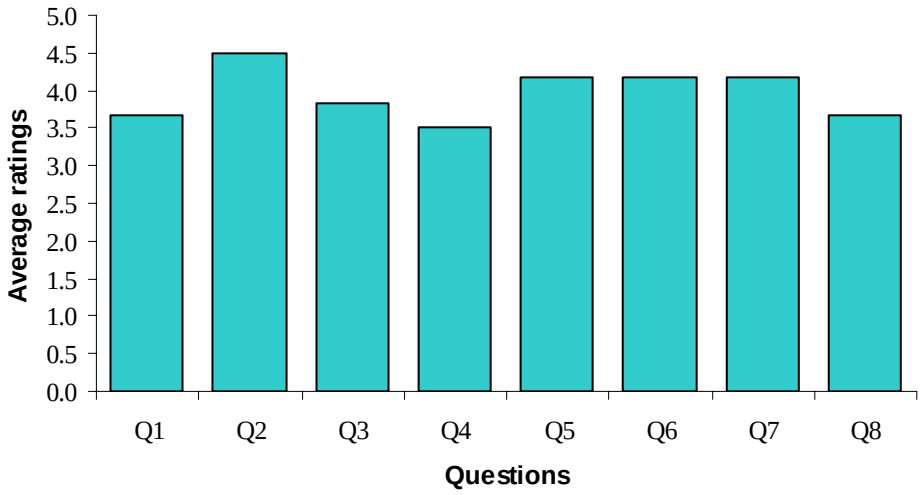


Figure 5.1: Average score of each question in the post-study questionnaire

Integrated into Eclipse

Table 5.3 and Figure 5.1 show that the average score of Q7 “I found the various functions in this system were well integrated” is 4.17. This suggests that participants agreed with this statement. Our observations also indicated that participants seemed to be satisfied with the way of integrating the annotation tool into Eclipse as they had no difficulties in conducting all tasks. However, in carrying out the task of editing the corresponding Java files according to the annotations, some participants failed to immediately realize that they had to return to the code window to change the program code, although they did eventually work it out. This problem could be reduced by allowing users to edit the code in the annotation window, namely, making the code in the annotation window editable.

Making free-form annotations

Our observations indicated that all participants encountered difficulty in writing dots and words smoothly and clearly but writing symbols presented no problem. A dot is an ink stroke that the pen touches the screen and is immediately off the screen without any drag. Participants invariably required several attempts to complete a dot, and were sometimes unable to one even after several attempts. Participants thus experienced stress and difficulty in writing comments. For example the word “input”, when the user goes back to dot the i after writing the word the dot may be invisible despite the user clearly having attempted to create it (Figure 5.1a).

To eliminate the difficulty in making dots, CodeAnnotator was reprogrammed to allow automatic completion of a dot when a user writes a dot. This was achieved by increasing the size of the dot to equal the line width of the pen or highlighter being used for writing (See Figure 5.1 b and c).

a **b** **c**

Figure 5.2: Making a dot. (a) the dot is invisible in original CodeAnnotator; (b) the dot has pen width after re-programming; (c) the dot has highlighter width after re-programming

The original CodeAnnotator was based on the assumption that users write a word using the minimum number of strokes. However, everyone has individual writing behavior. For example a word “check”, some individuals may write the word with one stroke, some may finish it with two, three, or four strokes, and others may write it letter by letter and complete it with five strokes. These different writing behaviors lead to different refreshing frequencies of the outline window. In the original CodeAnnotator, each time the user makes a stroke, the graphical list of annotations is refreshed and the group to which the stroke belongs is recognized, after which the recognized handwritten text is displayed in the graphical list. Particularly, recognizing a handwritten comment takes a certain amount of time because it is necessary to collect the points of all ink strokes of the comment each time and then recognize the comment according to these points. For example a sentence “check the input”, after completing “check the i”, the user writes “n” immediately followed by “p”, and then the user finds that “p” is hard to write. The factor that causes this problem is after writing “n”, the tool collects the ink points of “check the in” and recognizes the comment, and repaints the graphical copy of the comment, and then refreshes the graphical list of annotations in the outline window.

To improve performance in writing comments, handwriting recognition is performed only when users save the annotation window. In other words, before saving the annotation window, the outline window only contains a graphical list of existing annotations and gives no information on the text of the recognized handwritten comments. This considerably reduces the time required to refresh the graphical list of annotations in the outline window. For example, after the user writes “n”, the tool only repaints the graphical copy of the comment, after which “p” can be written immediately.

Editing annotations

Participants looked frustrated and stressed while making annotations. However, they looked relaxed and happy when editing existing annotations. Thus it seems that participants were happy with the performance of editing annotations in CodeAnnotator. The 4.5 average score for Q2 in Table 5.3 supports this by indicating that participants appeared satisfied with the performance of deleting, recoloring and moving existing annotations. In CodeAnnotator, existing annotations can easily be recolored by choosing a color from the color dialog, deleted with the eraser, and freely moved via the selector.

Navigation system

Table 5.3 shows that the average rating of Q3 -- “The navigation support aided me seeking and reviewing annotations” – is 3.83, indicating that participants were strongly apt to agree this view. One participant observed that "comments in the graphical list of annotations are hard to read”. In the Navigator window, a simplified graphical list of annotations is displayed under each annotation file. The image of each comment is too small for the comment itself to be recognized. This is because the CodeAnnotator Navigator is an extension of Eclipse Navigator, which fixes the size of images displayed in Navigator. Additionally, in the Outline window, the size of the image of each comment is fixed to the same size. The consistent image size may be suitable for some comments but too small for others, resulting in some comments being illegible. For example the two comments “Good” and “Good to have it” have identical image sizes, but in the case of the latter this sixe is such that the comment becomes illegible. (See Figure 5.3) This phenomenon occurs because the CodeAnnotator Outline window extends GEF Outline window, which fixes the size of images to the size of the image of the most first stroke made by users.

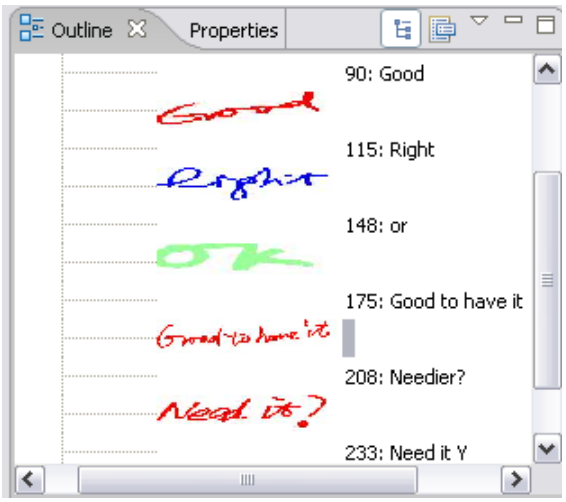


Figure 5.3: Images of comments in the Outline window

To improve this difficulty, two approaches can be adopted. First, a more accurate handwriting recognition system capable of reflecting the true meaning of handwritten comments could be employed. However, this approach would prelude the recording of pictorial comments. But users could just look at the recognized comment text to know what the comment is about. The alternative approach would be to make the image size flexible. This would require developers

thoroughly understanding Eclipse and digging deep into its base code to reprogram the way of painting images in the Navigator and Outline windows.

Grouping, anchoring and reflowing annotations

	Participant1	Participant2	Participant3	Participant4	Participant5	Participant6	Total
Group correct	3	5	3	3	3	4	21
Group incorrect	2	0	2	2	2	1	9

Table 5.4: Frequency of correctly grouping strokes

	Participant1	Participant2	Participant3	Participant4	Participant5	Participant6	Total
Anchor correct	3	5	3	3	3	4	21
Anchor incorrect	2	0	2	2	2	1	9

Table 5.5: Frequency of correctly anchoring groups

All participants wrote five annotations on the same Java code file, meaning the six participants made a total of 30 annotations. Table 5.4 shows that nine out of 30 annotations failed to be grouped correctly. In other words, around 77% of annotations were grouped correctly. The same situation exists for anchoring groups (Table 5.5). Incorrectly grouped strokes lead to improperly anchored groups. Incorrect grouping of strokes resulted from differences in participant writing behavior. The above is demonstrated by the following three situations.

- First, the current stroke overlaps the previous stroke horizontally, but the distance between them exceeds 10 pixels. For example a line linker and the comment “Fantastic”, after making the line linker, the user goes write “Fantastic” below the line linker, although the linker and “Fantastic” overlap horizontally, the distance between them are greater than 10 pixels. (See Figure 5.4) The linker and comment belong to different two groups. Additionally, the linker is associated with code line number 89, while the comment is anchored to code line number 93. In fact, both the linker and the comment should be anchored to code line number 89. To prevent this problem, the permitted

distance between the current and previous strokes is increased from 10 pixels to 35 pixels.

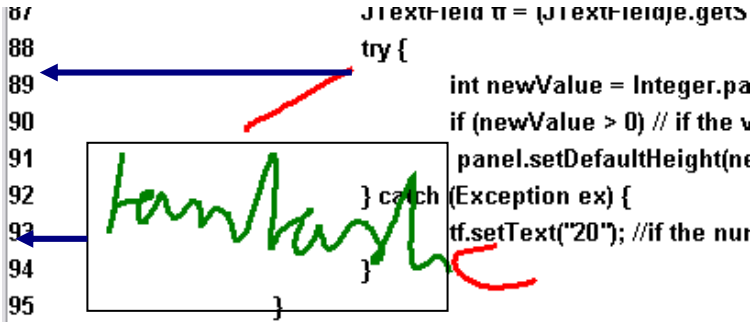


Figure 5.4: Stoke overlaps groups horizontally but the distance is greater than 10 pixels.

- Second, the current stroke does not overlap the previous stroke, but they should be in the same group. For example a line linker and the comment "come from where?", after making the line linker, the user goes to write "come" above the line linker, meaning the linker does not overlap "come", but they should be in the same group. (See Figure 5.5) The linker and comment belong to two groups respectively, causing them to be anchored to different code: the linker is anchored to code line number 83, while the comment is anchored to code line number 80. To overcome this problem, an additional grouping condition is used: as long as the horizontal distance between the current and previous strokes is less than 100 pixels and their vertical distance is less than 35 pixels, they are grouped together.

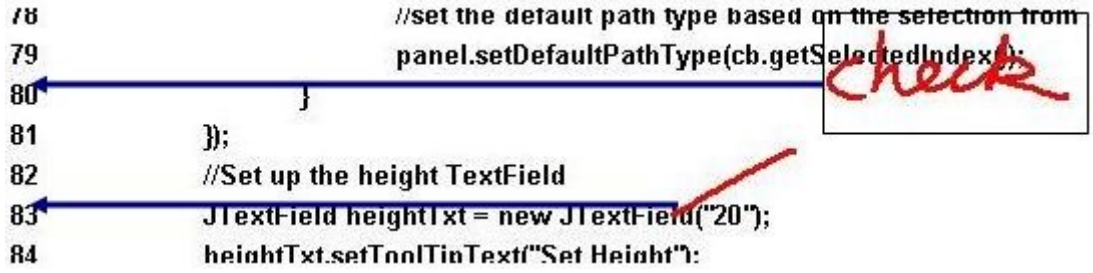


Figure 5.5: Stokes should be in the same group but they don't overlap.

- Third, a participant used a pair of brackets "()" as a linker enclosing several code lines (See Figure 5.6). Since the left bracket was distant from the right bracket, the two were considered to belong to different groups by the original and revised grouping

approaches, and thus were associated with different code. The current tool only allows linkers made with a single stroke, but a future tool could allow users to create linkers made with multiple strokes.

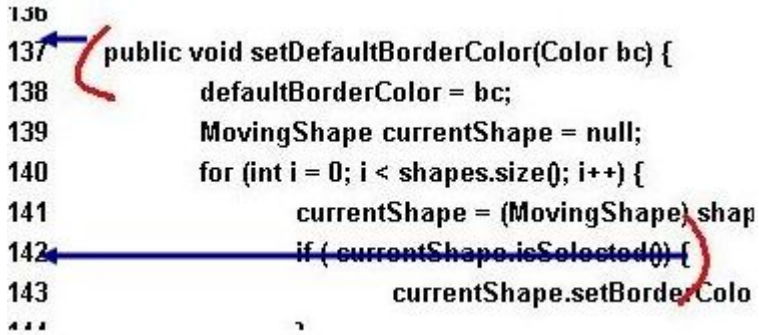


Figure 5.6: A linker is made in two strokes.

Most annotations made by participants could maintain their consistency with the associated code when the code document was edited. The accuracy rate of reflowing annotations was similar to that of grouping strokes and anchoring groups. No correctly grouped strokes were anchored improperly and lost their relationship with the attached code. However, incorrectly grouped strokes normally failed to anchor and reflow correctly. An accurate grouping approach could increase the accuracy of anchoring and reflowing annotations.

5.2 Summary

This chapter described the usability study that assessed the effectiveness of CodeAnnotator and identified its strengths and weaknesses. This study explored the weaknesses of the tool and generated some solutions to those weaknesses, as well as some ideas for improving or eliminating them. The next chapter represents an overall view of the implementation of CodeAnnotator, as well as related experiences and issues.

Chapter 6

Discussion

This thesis set out to explore an electronic environment that allows users to freely make digital ink annotations on digital code documents inside an IDE. CodeAnnotator was designed and developed to meet this requirement. This chapter discusses the influences on the design of this tool (the design itself was described in Chapter 3) and issues with the implementation of the tool (the implementation was presented in Chapter 4). After that, this chapter also discusses the findings obtained in the usability study to evaluate the effectiveness of the tool. (The usability study was described in Chapter 5) Finally, an overall summary of the tool is given.

6.1 Design and implementation discussion

CodeAnnotator is a set of plug-ins for Eclipse that is a developing platform for Java. Seamlessly integrating CodeAnnotator into Eclipse was a big challenge during development. Priest and Plimmer (2006) noted that it is difficult to seamlessly integrate an annotation tool into an IDE because of the limitations of the extensibility of the IDE. The desired goal was to allow users to review, edit, debug, and annotate directly in the same window – the code window. Three approaches for achieving a similar function were attempted: the first approach was to build a transparent window totally overlapping the code window to allow the two windows to act as one; the second approach was to build a separate window similar to the code window and allow users to perform all actions in this separate window; the third approach was to build a separate window (the annotation window) to hold ink annotations and allow users to review and annotate in the annotation window but edit and debug in the code window. The first two approaches failed, and thus the third approach was used to integrate CodeAnnotator into Eclipse. Various functions were then added to enrich the tool. Since the tool is integrated into Eclipse, it supports Java documents and allows users to edit documents being annotated.

With the pen and paper approach, there are no restrictions on annotation type or location. Marshall (1997) found that one of the critical aspects in assessing the quality of an annotation

tool is to allow annotations on digital files that are as rich in form as markings on paper. To permit this an annotation tool must enable users to make free-form annotations without restrictions of location. CodeAnnotator has this ability to fulfill this requirement. CodeAnnotator provides users with a digital environment for making free-form annotations within Eclipse. Additionally, users can easily modify existing annotations, such as users can delete, recolor and resize annotations without leaving any undesirable ink traces, and move annotations anywhere within the document.

Traditional ink annotations are made at specific locations on a page. Annotators do not need to worry about the meaning of annotations as they are affixed to their underlying content and paper documents are not editable. However, physical position in a digital file loses meaning when the file is edited (Bargeron & Moscovich, 2003). In other words, when digital files are edited the physical position of annotations underlying content can change, making annotations meaningless if no remedial action is taken. Instead, annotations on digital documents must be associated to with their surrounding logical context, like a segment of nearby text, a figure, several words and so on (Golovchinsky & Denoue, 2002; Bargeron & Moscovich, 2003). To preserve the meaning of annotations, a digital ink annotation tool must be able to anchor annotations to a specific context. CodeAnnotator uses the linker concept (Priest & Plimmer, 2006) to link annotations to their underlying code. A linker is an ink stroke and can be any kind of symbol, such as a line, circle, and bracket etc. A linker builds a bridge between a comment and its attached code. Linkers are made before a comment is written and are attached to a specific code line according to their type; line linkers are attached to the code line closest to their starting point, while other linkers are associated with the code line closest to the middle point of the bounding box. Comments made after a linker belong to that linker.

However, a comment consists of several words or symbols and is generally unable to be completed in a single stroke. Therefore ink strokes must be grouped before being attached to their corresponding linker. Ink strokes can be grouped based on either their temporal or spatial properties (Golovchinsky & Denoue, 2002; Bargeron & Moscovich, 2003). CodeAnnotator adopts both these types of properties in performing the grouping. Two strokes belong to the same group provided the time between the current stroke being made and the previous stroke being completed is less than half a second and/or the current stroke is near the previous stroke. Otherwise, the current stroke belongs to the closet group.

After being anchored and grouped, annotations can maintain consistency with their associated content when the document is edited. Existing annotations and their associated contents maintain their consistency by reflowing together. CodeAnnotator reflows annotations based on their anchor information, including the line number and content of the attached code line.

Recognizing handwritten annotations is a challenge owing to variation among individuals in handwriting behavior. Consequently, a single comment can be represented by various ink traces. CodeAnnotator integrates the Microsoft Tablet PC handwriting recognition system using a Java interface component (Coyette et al., 2007) to support users in recognizing handwritten comments and enable the possible future extension, which is to add the functionality of sorting and searching annotations.

Program code documents are structured differently to text documents, being organized logically by classes and methods and having no pagination boundaries. Providing a navigation system containing thumbnails of all pages, like XLibris (Schilit et al., 1998), is inefficient. Furthermore, merely provide a list of annotations for a document, like TAS (Brush et al., 2001), is inadequate, since a program project normally contains more than one document. CodeAnnotator provides a navigation system that contains a Navigator and an Outline window. The navigator contains all of the annotation files in the project and each one has a simplified graphical list of annotations. Meanwhile, the Outline window contains a detailed graphical list of annotations in a program code document. Users thus can review and locate annotations easily.

6.2 Evaluation discussion

The usability study obtained positive results. Participants were able to edit the code while making annotations on it, and could effortlessly make free-form annotations. Also participants could easily delete, move and recolor existing annotations without creating a mess. Furthermore, the tool allowed annotations to maintain their meaning since they reflow together with their associated code when the code changes. In addition, participants could easily and conveniently review and locate annotations.

However, the usability study exposed five weaknesses of the tool, and possible solutions and improvements were proposed for each.

- First, some participants failed to immediately realize that the program code could only be edited in the code window. Making the code in the annotation window editable would be an approach to reduce this problem.
- Second, participants experienced difficulty in creating dots. The tool thus was reprogrammed to automatically enlarge the dot size to the width of the pen or highlighter while users make a dot.
- Third, participants encountered difficulty in writing comments smoothly. Handwriting recognition thus was disabled during comment writing and activated only when users saved the annotation window or placed the mouse over the annotation in the selection mode.
- Fourth, participants felt that the comments in the Outline window were illegible. We proposed two ideas to improve this shortcoming: integrating more accurate handwriting recognition or reprogramming to make the image size flexible.
- Finally, some strokes made failed to be grouped correctly. In response, the original spatial grouping situations were extended to include as many writing behaviors as possible. Furthermore, it was proposed to allow users to make linkers that are completed in more than one stroke (like brackets).

The following five key observations are offered based on the usability study.

1. Many participants appeared to have high expectations regarding the ability to make annotations without limitations of type or location. Participants did not care whether or not such annotations were allowed to be made. Participants even wanted to be able to create graffiti if they felt like it. They also expected the window to automatically enlarge to accommodate annotations made outside it.
2. Every participant had their own particular writing manners. Some wrote words in a single stroke, while some required several strokes to complete a word. This individual variability in writing style affects the refreshing frequency of the graphical list of annotations in the Outline window.

3. Participants seemed to pay little attention to write comments near their corresponding linkers. After making a linker, some participants wrote the comment immediately

afterwards, while others wrote it slightly above or below. This tendency meant that a wide range of situations needed to be considered when grouping annotations.

4. Some participants had no difficulty in editing the program code in the code window and making annotations in the annotation window. However, some failed to immediately realize that they had to edit the code in the code window, although they did finally work it out. Consequently, the best approach is to enable users to conduct both activities in the same window.

5. Many participants hardly used the graphical list of annotations in the Outline window to locate specific annotations. Instead they scrolled the screen up or down to find annotations. One participant said that “it is hard to read the comments in the list”. Thus it is important to provide a legible list of annotations.

However, CodeAnnotator did allow participants to make annotations inside the annotation window and edit the program code in the code window. Participants could also readily and easily modify existing annotations. Additionally, existing annotations reflowed to maintain their visual meaning and style. Furthermore, the navigation system helped participants find specific annotations. One participant commented that “the application was great”. Another commented that “it was a great system”. The overall feedback from participants was that the system performed exceptionally well and was both helpful and useful.

6.3 Summary

CodeAnnotator is a successful program code digital ink annotation tool that provides users with a pen-and-paper like environment for annotating program code files in an IDE. This tool includes all of the critical features an annotation tool is required to possess, such as free-form annotations, grouping and reflowing annotations. Additionally, this tool provides a handwriting recognition system for recognizing handwritten comments. Furthermore, it enables users to review and find annotations easily and conveniently through its navigation system.

The usability study explored the strengths and weaknesses of CodeAnnotator. Solutions and ideas for improvement were offered for each weakness to help overcome them. Five key

observations were also offered. The usability study also explored the strengths of the tool: allowing make annotations within the annotation window without constraints of annotation type or location, easily modify existing annotations, reflow annotations to maintain consistency with associated code, and receive assistance in reviewing and locating annotations.

The next chapter presents conclusions together with the contributions of this research. Some interesting directions for future extensions are also presented.

Chapter 7

Conclusion and Future Work

This thesis describes a program code digital ink annotation tool, CodeAnnotator. This tool is intended to provide users with an electronic environment for annotating in Eclipse. CodeAnnotator integrates digital ink annotation into Eclipse for reviewing and annotating Java code, and lets users enjoy both digital ink annotation and Eclipse support. In other words, after loading a code file in Eclipse, users can write annotations with a digital pen and can also access all the functions provided by Eclipse, such as debugging, compiling, and executing.

This thesis has contributed to the knowledge on developing program code digital ink annotation tools within an IDE. CodeAnnotator makes six notable contributions.

- First, CodeAnnotator is well integrated into Eclipse, thus allowing users to enjoy both Eclipse and annotation support.
- Second, CodeAnnotator enables users to directly annotate over code in Eclipse.
- Third, CodeAnnotator allows users to make annotations over code as they can on paper.
- Fourth, CodeAnnotator groups strokes and anchors each grouped annotation to a specific code, thus maintaining consistency between annotations and their associated content when the underlying code changes.
- Fifth, CodeAnnotator recognizes handwritten comments and converts them to a standard font.
- Finally, CodeAnnotator provides a navigation system to allow users to review and locate annotations easily and effectively.

In the short-term, future work could implement improvement ideas represented in the Evaluation chapter, including: making the program code in the annotation window editable, improving the legibility of annotations in the Outline window, integrating a more accurate handwriting recognition tool into CodeAnnotator, and allowing users to make linkers that are made in more than one stroke (for instance, brackets). Subsequent steps would involve

conducting a formal usability study to thoroughly evaluate CodeAnnotator, and performing a comparison study to test whether the tool is an easy and convenient way to make and review annotations in Eclipse, as well as to identify its advantages.

The next possible improvement would be to find a way to allow users to annotate directly over the program code. Users then could perform all actions (e.g. editing, reviewing, debugging, and annotating) in the code window. Another approach would be to make the annotation window editable. Users could change the code directly on the annotation window and these changes would reflect back to the code window automatically and simultaneously. This could possibly be achieved by changing the base code rather than extending with Plug-ins.

CodeAnnotator could also be extended to have the functionality to search and sort existing annotations. The current Outline window lists annotations in order of the code line number to which they are anchored. This facility could be extended to allow annotations to be sorted according to comments, and to provide a search engine to allow users to search for a specific annotation. These extensions require a handwriting recognition system that can accurately recognize handwritten comments.

Other interesting extension would be functionality to consolidate digital ink annotations made by different users into a single file to support collaborative code review. The separate digital ink annotations could be merged automatically by using some mechanism to differentiate annotations made by different individuals, particularly those sharing the same place. Furthermore, CodeAnnotator could be extended to support assignment marking. Markers then would be able to directly mark and comment on programming assignments in Eclipse.

Appendix 1

Usability Testing on CodeAnnotator

Project Name: Digital ink annotation in Eclipse
Name of Tester(s): Xiaofan Chen
Supervisor: Dr. Beryl Plimmer
Ethics Approval Number: 2007/383

Overview:

This project aims to provide users with an electronic environment to annotate code documents within an Integrated Development Environments (IDE).

Usability Test Type:

Assessment – measure the effectiveness of the implementation

Purpose:

Evaluate the strengths and limitations of CodeAnnotator.

Test Environment / Equipment:

The test environment consists of a room with a tablet computer, with keyboard covered, but wireless keyboard and mouse available and an observer chair. The tablet computer is set up with Eclipse and Morae Recorder. Monitoring the test and mouse movement and keystrokes is Morae Recorder.

Test Objectives:

Function	***Problem statements***
Make comments	*1. Are the users able to find pens/highlighters with different colors?* *2. Are the users able to make annotations any where inside the annotation window?* *3. Are the users able to make free-form annotations?* *4. Do the users feel difficult to change existing annotations (recolor, delete, move)?*
Group comments	*1. Does the group function implement correct?*
Reflow comments	*1. Is the tool able to reflow existing annotations correct when the underlying contents are changed?*
Navigation support	*1. Does the navigation system aid or hinder users seeking/reviewing annotations?*

Profile:

Participants selected for this research will come form the following groups: students, markers and teachers. All participants will have some experience with programming Java and using an IDE such as Visual Studio, Eclipse.

M**ethodology (Task Design):**

The test will consist of a performance test made of three sections:

- *Participant greeting and background questionnaire*

Each participant will be personally greeted by the tester.
Particpants will be given the participant information sheet and ethical approval form and asked to read and sign the form.
The participants will be asked to fill out a short pre-study questionnaire which gathers background information.

- *Training*
  Training participants how to make annotations, edit existing annotations, group and reflow existing annotations, and use navigation support. Then given a series of practice tasks

- *Performance test*
  The performance test consists of a series of tasks that the participants will carry out while being observed.
    - *Participants will be handed a script, detailing the tasks to be accomplished. They will then attempt the tasks.*
    - *After concluding the tasks, the participants will complete a short survey to gather their responses to the tasks and the overall comments and recommendations to the tool*

Pilot Testing:

The pilot test will be conducted at 2pm on 30 Nov. 2007.This is to test out the usability tasks and questions to determine if they are suitable (Question/Tasks may be too difficult/easy/long/short) and to make changes if they aren't.

Acquiring Participants:

Can be done via email/invitation notices/word of mouth. – check the ethics
Need to determine the number of participants required for your test.

Training Script:

1. *Open a Java file.*
2. *Review this Java file, you find somewhere you want to make comments.*
3. *Click 'Pen' button on the Toolbar of Eclipse.*
4. *An annotation window with the content of the Java file will be opened.*
5. *Select a colored pen/highlighter on the left palette of the annotation window.*
6. *We have to make a linker before writing comments. A linker is linking the comment to a specific code line.*
7. *A linker can be a straight line, a circle, a square, or anything you like*
   a. *For a straight line, the attached Java line is determined by its first point*
   b. *For others, the attached Java line is the middle of the height of the linker*
8. *After making the linker, we can write comments.*
9. *If you want to delete a comment, click 'Erase' button on the left palette of the annotation window, and click the comment you want to delete. Then the comment and its linker are erased.*

10. *If you want to move a comment to a new place, click 'Select' button, and select the comment and drag it to a new place. Then the comment and its linker are moved, and its attached java line is changed too.*
11. *If you want to emphasize a comment, click 'Color' button, select a color, and click the comment. Then the comment is recolored.*
12. *Click 'Save' button on the Toolbar of Eclipse to save the work.*
13. *After finishing making annotations, you find there is somewhere in the Java file you want to change, such as adding more lines, deleting some lines, modifying some classes. Go to Java file window, change things you want to change. Then go back to the annotation window, all existing annotations are kept attached to the appropriate line (reflow is 'super-geek' speak that they will not understand).*
14. *Save the work and close the annotation window.*
15. *If you modify the Java file after closing the annotation window. Next time you open its annotation file, all existing annotations are reloaded.*
16. *We built a navigation system to aid users to review and find annotations. The navigation system consists of a CodeAnnotator Navigator and an Outline window.*
    a. *The CodeAnnotator Navigator is based on the file level. From here, you can find which Java files have their annotation files. Expanding an annotation file, you can see a simplified graphic list of its existing annotations.*
    b. *Open an annotation file you want to review.*
    c. *At the same time, an Outline window is opened. In the Outline window, there is a detailed graphic list of the existing annotation. Select a comment in the list, the annotation window will scroll automatically to the location of the commen*

Task List:

Task 1	**Description**
Task	*Open a new annotation window for a Java file*
Benchmark	*1 minute*
Script	*Imagine you are marking a student's assignment, you want to write some comments on the student's code files.* *From the Navigator window to open A1.java* *Review this Java file, Then click the 'pen' button in the eclipse toolbar to open a new annotation window for the Java file*

Task 2	**Description**
Task	*Write 3 or 5 comments*
Benchmark	*3 minute*
Script	*Review A1.java, and write 3 or 5 comments on the annotation window.* ***When reviewing method "setUpToolsPanel()", you note that this method calls createImageIcon(). As you don't find out a method called createImageIcon(), write a comment to the student about "Come from where?"*** *(Choose a red pen from the left palette of the annotation window, go to line 57, make a straight line in line 57, and write "Come from where")* ***One of the criteria for the assignment is try/catch. Find the try/catch and provide positive reinforcement for doing it right, could be a ☺or tick or write "Fantastic"*** *(Go to line 88, make a circle between line 88 and 94, write "Fantastic".)* ***One of the criteria for the assignment is to check the inputs of textFields to make sure all textFields have correct inputs.For the width textField, the input has to be interger, but the student fails to check the input of this textField. Write a comment to the student about "Check the input"*** *(Go to line 98, make a big bracket across line 98 to line 104, and write "Check the input")* ***When reviewing method "setUpButtons()", you note that there are two startButton.setToolTipText(), write a comment to the student to change one of them to stopButton.setToolTipText().*** *(Choose a yellow highlighter from the left palette of the annotation window, go to line 189, highlight line 189, and write "Change to stopButton")* ***After reviewing the code, you find that the student made a great job; you want to give an overview comment "Well-done" at the***

	end of the code. *(Go to the end, write "Well-done")*

Task 3	**Description**
Task	*Delete, recolor, move an existing comment*
Benchmark	*3 minute*
Script	*After making comments, you want to delete some comments, or recolor them, or move the comment to a new place.* ***Does createImageIcon() exist? If so, one of comments can be deleted.*** *(Click the 'Erase' button from the left palette of the annotation window, then go to line 57, delete the comment "Where does it come from?")* ***As the student made a great job in try/catch, you want to emphasize the comment's color to pink.*** *(Click the 'Color' button from the left palette of the annotation window, choose 'pink' from color window, then go to line 88, and recolor the comment in line 88, "Fantastic".)* ***It is better to put the overview comment "Well-done" to the beginning of the code.*** *(Click the 'Select' button from the left palette of the annotation window, then go to the end, move "Well-done" to the beginning.)*

Task 4	**Description**
Task	*Append words to an existing comment*
Benchmark	*1 minute*
Script	*After making several comments, you find a comment is incomplete and want to add more.* ***Using a blue pen to append "use If conditions" to "Check the input".*** *(Choose a blue pen from the left palette of the annotation window. Go to line 98, add "use If conditions" after the "Check the input".)* ***Save this annotation file. And close it and the Java file.***

Task 5	**Description**
Task	*Navigate the existing annotation files to find a specific comment and according to the comment to change the Java code.*
Benchmark	*3 minute*
Script	*Normally, a Java project consists of several Java files, then*

	probably each Java file has an annotation file. You want to review the existing annotation files and locate a specific comment. ***First**, find the comment "Delete these, keep code clean" in MovingShape.ink. Then according to the comment to delete lines in MovingShape.java. Check whether this comment is deleted and all other annotations in MovingShape.ink are reflowed.* ***Second**, find the comment "change to Vectore<MovingShape>" in AnimationPanel.ink. Then according to the comment to change the Vector in AnimationPanel.java. Then check whether this comment in AnimationPanel.ink is deleted and other comments are reflowed.*

Actual Test

Test Preparation (Training):

- *read participant information sheet*
- *sign ethics approval*
- *Pre-test questionnaire.*
- *User Training*
    - *Explain what the project is about and what the software does (background knowledge)*
    - *Getting the user familiar with the equipment*
    - *Demonstrate the functions of the software with an example.*
    - *You may want to train the user in sections so as to not overload the user with excessive information*

Actual Testing:

Check the recording devices are turned on

Ask participants to complete particular tasks.

Post Test

- *Post-test questionnaire / informal interview*
    - *This is where qualitative information is acquired such as how easy it is to use the software, how easy the tasks were.*
    - *The questionnaire may also be filled out after each scenario.*
- *Thank the participant – hand out reward/thank you*

Evaluation Analysis:

The measures that will be used to evaluate the application are as follows:

1. *Quantitative:*
    - *Time required completing the task*
    - *Count of incomplete tasks*

2. *Qualitative:*
    - *User emotions during the test (ie frustration/irritation, confusion)*
    - *User feedback from questionnaire*
    - *User comments and mannerisms*

Sample Advertisement/ email/notice

Department of Computer Science
The University of Auckland
Private Bag 92019
Auckland

Tel: 09 373 7599

My name is Dr. Beryl Plimmer, I am a Senior Lecturer in the Department of Computer Science at the University of Auckland. I am conducting research into pen input computing working with Xiaofan Chen, who is under my supervision. I am investigating how computers can support more natural computer interaction. A part of exploring these ideas is involving potential 'ordinary' users in the design, usability testing and evaluation of the prototype applications. This particular project aims to support digital ink annotation in software development tools.

In this study we are testing the usability of a prototype system for *CodeAnnotator*. The studies are conducted in the HCI Lab in Computer Science or your office and will take a maximum of 40 minutes.

You are invited to participate in our research and we would appreciate any assistance you can offer us, although you are under no obligation to do so. The studies will take place between 01/12/2007 and 31/12/2007. If you would like to participate please email her to arrange a time, xche044@ec.auckland.ac.nz

Regards

Dr. Beryl Plimmer

This research has been approved by The University Of Auckland Human Participants Ethics committee on 14 November 2007 for a period of 3 years from 14 November 2007. Reference 2007/383 .

Department of Computer Science
The University of Auckland
Private Bag 92019
Auckland

Tel: 09 373 7599

PARTICIPANT INFORMATION SHEET

Title: Digital pen input for computer applications

To participants:

My name is Dr. Beryl Plimmer, I am a Senior Lecturer in the Department of Computer Science at the University of Auckland. I am conducting research into pen input computing with Xiaofan Chen. I am investigating how computers can support more natural computer interaction. A part of exploring these ideas is involving potential 'ordinary' users in the design, usability testing and evaluation of the prototype applications. This particular project aims to support digital ink annotation in software development tools.

In this study we are testing the usability of a prototype system for *CodeAnnotator*.

You are invited to participate in our research and we would appreciate any assistance you can offer us, although you are under no obligation to do so.

Participation involves one visit to our laboratory at The University of Auckland, for approximately 40 minutes, and requires that your eyesight is normal or corrected-to-normal by spectacles or contact lenses. If you agree to participate, you may be asked to perform a number of tasks on paper, using a computer via the keyboard or mouse and using a computer with a pen. The tasks will be fully explained and demonstrated. You will be asked to annotate documents and review annotated documents. The changes you make and the time you spend working on each task will be digitally recorded together with synchronized video. You will be asked fill in a short questionnaire to note your age, education level and existing experience with the tasks and technology and complete a short questionnaire on your experience.

All the questionnaire information you provide will remain anonymous. The digital recordings, with your specific consent, may be used in research reports on this project. You choose whether your recordings are used or not on the consent form. Your consent form will be held in a secure file for 6 years, at the end of this time it will be properly disposed of. Your name will not be used in any reports arising from this study. The information collected during this study may be used in future analysis and publications and will be kept indefinitely. When it is no longer required all copies of the data will be destroyed. At the conclusion of the study, a summary of the findings will be available from the researchers upon request.

If you don't want to participate, you don't have to give any reason for your decision. If you do participate, you may withdraw at any time during the session and you can also ask for the information you have provided to be withdrawn at any time until one week after the conclusion of your session, without explanation and without penalty, by contacting me (details below). If you are a student at The University of Auckland choosing not to participate, or to

withdraw yourself or your information, your grades or academic relationships with the University or members of staff will not be affected.

If you agree to participate in this study, please first complete the consent form attached to this information sheet. Your consent form will be kept separately from your questionnaire data so that no-one will be able to identify your answers from the information you provide.

This project is partly supported by funding from Microsoft Research Asia and University of Auckland Faculty of Science Research Fund project number 360933/9343

Thank you very much for your time and help in making this study possible. If you have any questions at any time you can phone me (3737599 ext 82285) or the Head of Department, Associate Professor Robert Amor (3737599 ext 83068), or you can write to us at:

Department of Computer Science,
The University of Auckland
Private Bag 92019
Auckland.

For any queries regarding ethical concerns, please contact The Chair, The University of Auckland Human Participants Ethics Committee, The University of Auckland, Private Bag 92019, Auckland. Tel. 3737599 ext 87830.

APPROVED BY THE UNIVERSITY OF AUCKLAND HUMAN PARTICIPANTS ETHICS COMMITTEE on 14 November 2007 for a period of 3 years from 14 November 2007. Reference 2007/383 .

Department of Computer Science
The University of Auckland
Private Bag 92019
Auckland

Tel: 09 373 7599

CONSENT FORM

This consent form will be held for a period of at least six years

Title: Investigation of gesture input computing

Researcher: Dr. Beryl Plimmer

I have been given and understood an explanation of this research project. I have had an opportunity to ask questions and have them answered. I understand that at the conclusion of the study, a summary of the findings will be available from the researchers upon request.

I understand that the data collected from the study will be held indefinitely and may be used in future analysis.

I understand that I may withdraw myself and any information traceable to me at any time up to one week after the completion of this session without giving a reason, and without any penalty.

I understand that I may withdraw my participation during the laboratory session at any time.

I understand that my grades and relationships with The University of Auckland will be unaffected whether or not I participate in this study or withdraw my participation during it.

I agree to take part in this research by completing the laboratory session.

I confirm that my eyesight is normal or corrected-to-normal.

I **agree/do not agree** digital and video recordings taken during the session being used research reports on this project.

Signed:

Name:

(please print clearly)

Date:

APPROVED BY THE UNIVERSITY OF AUCKLAND HUMAN PARTICIPANTS ETHICS COMMITTEE on 14 November 2007 for a period of 3 years from 14 November 2007. Reference 2007/383 .

Questionnaire

Complete only This Section before the session

Your gender	Male	Female			
Your current university status	Student	Marker	Teacher		
How much experience do you have in programming Java	Less than 1 year	Between 1 and 3 years	Between 3 and 5 years	More than 5 years	
How much experience do you have on IDE (e.g. Visual studio, Eclipse)	Less than 1 year	Between 1 and 3 years	Between 3 and 5 years	More than 5 years	
I use an IDE for programming Java	Always	Usually	Sometimes	Rarely	Never
I use a digital pen for marking/making comments on digital files	Always	Usually	Sometimes	Rarely	Never
I use computer tools for making comments on digital files	Always	Usually	Sometimes	Rarely	Never
I have used a pen input on a computer	Frequently	Occasionally	A couple of times	Once	Never

---

The following sections will be completed after the session

General Questions

	Strongly Agree	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
It was easy to make annotations					
It is easy to change existing comments (recolor, delete and move)					
The navigation support aided me seeking and reviewing comments					
This tool is helpful and useful for me to make comments on code files					
This exercise was enjoyable					
I thought that this system was easy to use					
I found the various functions in this system were well integrated.					
I would like to use this method of interaction in the future					

Comments/Recommendations:

Bibliography

Adler, A., Gujar, A., Harrison, B.L., O'Hara, K. and Sellen, A. (1998). A diary study of work-related reading: design implications for digital reading devices. *Proc. CHI'98*, pp. 241-248

Anderson, R., Anderson, R., Simon, B., Wolfman, S.A., VanDeGrift, T. and Yasuhara, K. (2004). Experiences with a Tablet PC based lecture presentation system in Computer Science courses. *Proc. of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, pp. 56-60

Aniszczyk, C. (2005). *Using GEF with EMF*. IBM. Retrieved 26/01/2008 from http://www.eclipse.org/articles/Article-GEF-EMF/gef-emf.html

Aeppli, B. (2005). *Eclipse-based front-end for OMSjp*. Diploma Thesis. Global Information Systems Group, Department of Computer Science, ETH Zurich. Retrieved 09/01/2008 from http://e-collection.ethbib.ethz.ch/ecol-pool/dipl/dipl_151.pdf

Bargeron, D. and Moscovich, T. (2003). Reflowing digital ink annotations. *Proc. CHI 2003*, April 5-10, 2003, vol. 5, Issue 1, pp. 385-392

Ben-Ari, M. (2001). Constructivism in computer science education. *Journal of Computers in Mathematics & Science Teaching*, 20(1), pp. 24-73

Bottoni, P., Labella, A., Trinchese, R. and Gigli, L. (2006). MADCOW: a visual interface for annotating web pages. *Proc. of AVI'06*, Venezia, Italy, pp. 314-317

Bretzing, B.H. and Kulhavy, R.W. (1998). Note-taking and passage style. *Journal of Educational Psychology*, vol. 73, pp. 242-250

Brush, A.J.B., Bargeron, D., Gupta, A. and Cadiz, JJ (2001). Robust annotation positioning in digital documents. *Proc. of SIGCHI'01*, Seattle, WA, USA, vol. 3, Issue 1, pp. 285-292

Cadiz, JJ, Grpta, A. and Grudin, J. (2000). Using web annotations for asynchronous collaboration around documents. *Proc. of CSCW'00*, Philadelphia, PA, pp. 309-318

Chiu, P. and Wilcox, L. (1998). A dynamic grouping technique for ink and audio notes. *Proc. of UIST'98 Annual symposium on User Interface Software and Technology*, San Francisco, California, pp. 195-202

Clayberg, E. and Rubel, D. (2006). *Eclipse: building commercial-quality plug-ins (2nd edition)*. Pearson Education, Inc. ISBN 0-321-42672-X

Colen, L. (2001). Code Reviews. *Linux Journal*, Volume 2001, Issue 81es, Article No. 11.

Crawford, W. (1998). Paper persists: why physical library collections still matter. *Online Magazine*, January/February, pp. 42-48

Coyette, A., Kieffer, S. and Vanderdonckt, J. (2007). Multi-Fidelity prototyping of user interfaces. *In Interact 2007*. Brazil, pp. 150-164

Dumas, J. and Redish, C. (1999). *A practical guide to usability testing (Revised Edition)*. Intellect Books, pp. 21-39, ISBN 1841500208.

Ellis, S.E. and Groth, D.P. (2004). A collaborative annotation system for data visualization. *Proc. of AVI'04*, Gallipoli, Italy, pp. 411-414

Flanagan, D. (2001). *JavaScript: the definitive guide (4th Edition)*. O'Reilly, NY 2001

Fowler, R.L. and Barker, A.S. (1974). Effectiveness of highlighting for retention of text material. *Journal of Applied Psychology*, vol. 59, pp. 358-364

Francik, E.R. (1995). Rapid, integrated design of a multimedia communication system. *Human Computer Interface Design (1995)*, pp. 36-69

Golovchinsky, G. and Denoue, L. (2002). Moving Markup: Repositioning freeform annotations. *Proc. UIST 2002*, ACM Press (2002), 21-30

Hamzah, M.D., Tano, S., Iwata, M. and Hashiyama, T. (2006). Effectiveness of annotating by hand for non-alphabetical languages. *Proc. of CHI 2006*, Montreal, Canada, pp. 841-850

Heinrich, E., and Lawn, A. (2004). Onscreen marking support for formative assessment. *Proc. Of the World Conference on Educational Multimedia Hypermedia & Telecommunications*, Norfolk, pp. 1985-1992

Huang, A., Doeppner, T. and Cetintemel, U. (2003). *Ad-hoc collaborative document annotation on a Tablet PC*. Retrieved 12/01/2008 from http://www.cs.brown.edu/research/pubs/theses/ugrad/2003/ashuang.pdf

Koga, T., Tashiro, N., Ozono, T., Ito, T. and Shintani, T. (2005). Web Page Marker: a web browsing support system based on marking and anchoring. *Proc. of WWW 2005*, Chiba, Japan, pp. 1012-1013

Lee, D. (2003). *Display a UML diagram using Draw2D*. IBM. Retrieved 26/01/2008 from http://www.eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html

Marshall, C.C. (1997). Annotation: from paper books to the digital library. *Proc. Digital Libraries 1997*, ACM Press (1997), pp. 131-140

Marshall, C.C. (1998). Toward an ecology of hypertext annotation. *Proc. HyperText 1998*, ACM Press (1998), pp. 40-48

Marshall, C.C. (2003). Reading and interactivity in the digital library: creating an experience that transcends paper. *Proc. of the CLIR/Kanazawa Institute of Technology Roundtable*, Kanazawa, 5(4), pp. 1-20

Marshall, C.C., and Brush, A.J.B. (2002). From personal to shared annotations. *Proc. of CHI'02*, pp. 812-813

Marshall, C.C., and Brush, A.J.B. (2004). Exploring the relationship between personal and public annotations. *Proc. of the 4th ACM/IEEE-CS Joint Conference on Digital Libraries*, Tuscon, Arizona, pp. 349-357

Marshall, C.C., Price, M.N., Golovchinsky, G. and Schilit, B.N. (1999). Introducing a digital library reading appliance into a reading group. *Proc. ACM Digital Libraries 1999*, pp. 77-84

Melhem, W. and Glozic, D. (2003). *PDE does Plug-ins*. IBM Canada Ltd. Retrieved 27/01/2008 from http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html

Microsoft Office Online (2008). *Microsoft Office XP pack for Tablet PC (Tablet Pack)*. Retrieved 20/01/2008 from http://office.microsoft.com/downloads/

Microsoft Visual Studio (2005). *Visual Studio 2005*. Retrieved 21/01/2008 from http://msdn.microsoft.com/vstudio

Mock, K. (2004). Teaching with Tablet PC's. *Journal of Computing Science in Colleges*, 20(2), pp. 17-27

O'Hara, K., and Sellen, A. (1997). A comparison of reading paper and on-line documents. *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, Atlanta, Georgia, pp. 335-342

Olsen Jr. D.R., Taufer, T. and Fails, J.A. (2004). ScreenCrayons: annotating anything. *Proc. UIST'04*, vol. 6, Issue 2, pp. 165-174

Ovsiannikov, I.A., Arbib, M.A., and McNeil, T.H. (1999). Annotation technology. *International Journal of Human-Computer Studies*, vol. 50, pp. 329-362

Oostendorp, H. van (1996). Studying and annotationg eclectronic text. In *Hypertext and Cognition*, J. Rouet, J.J. Levonen, A. Dillon, R.J. Spiro, Eds. Mahway, NJ: Lawrence Erlbaum Associates, 1996, pp. 137-147

Plamondon, R. and Srihari, S.N. (2000). On-line and Off-line handwriting recognition: a comprehensive survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 22(1), pp. 63-84

Plimmer, B. and Apperley, M. (2007). Making paperless work. *Proc. CHINZ 2007*, Hamilton, New Zealand, pp. 1-8

Plimmer, B. and Mason, P. (2006). A Pen-based paperless environment for annotating and marking student assignments. *Proc. 7th Australasian User Interface Conference*, CRPIT press (2006), 37-44

Phelps, T.A. and Wilensky, R. (1996). Towards active, extensible, networked documents: multivalent architecture and applications. *Proc. of Digital Libraries'96*, pp. 100-108

Priest, R. and Plimmer, B. (2006) RCA: Experiences with an IDE annotation tool. *Proc. 6th ACM SIGCHI New Zealand chapters international conference 2006*, ACM Press (2006), 53-60

Priest, R.A. (2006). *Ink annotation for programming environments*. Master thesis, University of Auckland, New Zealand.

Ramachandran, S. and Kashi, R. (2003). An architecture for Ink Annotations on Web Documents. *Proc. of the 7th International Conference on Document Analysis and Recognition*, pp. 256-260

Redmond, W. (2002). *Q&A: Tablet PC hardware makers sought flexibility, durability, ultra-portability for "Corridor Warriors"*. Retrieved 19/01/2008 from http://www.microsoft.com/presspass/features/2002/nov02/11-05TabletPCpartners.mspx

Schilit, B.N., Golovchinsky, G. and Price, M.N. (1998). Beyond paper: supporting active reading with free form digital ink annotaitons. *Proc. CHI'98*, pp. 249-256

Schumacher, G.M. and Nash, J.G. (1991). Conceptualizing and measuring knowledge change due to writing. *Research in the Teaching of English*, vol. 25, pp. 67-96

Shipman, F.M., Price, M.N., Marshall, C.C., Golovchinsky, G., and Schilit, B.N. (2003). Identifying useful passages in documents based on annotation patterns. *Proc. of ECDL'03*, Springer Verlag, Heidelberg, pp. 101-112

Windows History (2002). *Windows history: windows desktop products histor*y. Retrieved 19/01/2008 from http://www.microsoft.com/windows/WinHistoryDesktop.mspx

Wolf, J.L. (2000). Effects of annotations on student readers and writers. *Digital Libraries*, San Antonio, TX, pp. 19-26

Zoio, P. (2004). *Building a database schema diagram editor with GEF*. Realsolve Solutions Ltd. Retrieved 26/01/2008 from http://www.eclipse.org/articles/Article-GEF-editor/gef-schema-editor.html