The University of Auckland

Thesis Consent Form

This thesis may be consulted for the purpose of research or private study provided that due acknowledgement is made where appropriate and that the author's permission is obtained before any material from the thesis is published.

I agree that the University of Auckland Library may make a copy of this thesis for supply to the collection of another prescribed library on request from that Library; and

1. I agree that this thesis may be photocopied for supply to any person in accordance with the provisions of Section 56 of the Copyright Act 1994

Or

2. ~~This thesis may not be photocopied other than to supply a copy for the collection of another prescribed library~~

(Strike out 1 or 2)

Signed:……………………………………………………………….

Date:…………

INK ANNOTATION
FOR
PROGRAMMING ENVIRONMENTS

Richard Anthony Priest

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

August 2006

ABSTRACT

Ink annotation is a popular approach for recording feedback over text documents, either on paper or over a digital document. However, we are interested whether this success can be seen when annotating program code directly within an Integrated Development Environment (IDE).

Program code is different from text documents in that it is non-linear and therefore it can become inefficient to annotate over printed versions of code, as the reviewer must manually search through pages of code to find method declarations and object classes. Reading code within an IDE, then documenting code issues on a piece of paper or in a text editor involves frequent switching between the code and the issue list, risking a break in concentration, while also giving no direct reference point to the code, resulting in each issue requiring a higher degree of description. Lastly, coding issues should not be written directly within the IDE as a comment, as they can be difficult to distinguish from actual code comments. The ability of digital ink annotation directly in an IDE gives the reviewer a more distinguishable and intuitive annotation approach, and gives a computerized approach to improve efficiency problems seen in code review, while also retaining all the benefits an underlying code editor has to offer.

In this thesis we investigate the necessary requirements for a digital ink annotation tool within the IDE, this will provide benefit to any scenario where one reads or writes code. Other than supplying ink over code, several annotation ideas and code review concepts are also investigated which contribute to our initial ink annotation prototype, 'Rich Code Annotation' (RCA). We present the implementation of the RCA tool and scrutinize our tool with two in-depth evaluations. Finally we discuss our final prototype and comment on the further work that can be incorporated in future versions of this tool.

ACKNOWLEDGEMENTS

First and foremost I want to strongly express my thanks and deepest gratitude to my supervisor, Beryl Plimmer. Thank you also to John Hosking and John Grundy, who provided some interesting comments throughout my Masters year.

Thanks you to my friends and family who offered encouragement and support throughout my academic years, this was much appreciated.

Finally, thanks to all those who evaluated my tool and offered improvements and further ideas.

TABLE OF CONTENTS

CHAPTER 1 – INTRODUCTION 1

1.1 GENERAL INTRODUCTION 1

1.2 OUR MOTIVATION 1

1.3 OUR GOAL 3

1.4 OUR APPROACH 3

1.5 THESIS OVERVIEW 4

1.6 SUMMARY 5

CHAPTER 2 – RELATED WORK 7

2.1 INTRODUCTION 7

2.2 INK ANNOTATION 7

2.2.1 Traditional ink annotation ……………………………………………………..8

2.2.2 Digital document annotation ……………………………………………………12

2.2.3 Digital ink annotation ………………………………………………………..13

2.2.4 Technical Challenges …………………………………………………………..17

2.3 CODE REVIEW 20

2.3.1 Walkthrough …….……………………………………………………….......21

2.3.2 Inspection ………………………………………………………………………22

2.3.3 M. E. Fagan ……………………………………………………………………..23

2.3.4 Walkthrough versus inspection ………………………………………………....27

2.3.5 Code review difficulties ………………………………………….…………..28

2.4 RELATED ANNOTATION TOOLS 32

2.5 RELATED INTEGRATION TOOLS 40

2.6 SUMMARY 45

CHAPTER 3 – SCENARIO OF USE 47
3.1 INTRODUCTION 47
3.2 TEACHING IN THE CLASSROOM 47
3.3 MARKING 48
3.4 FORMAL TECHNICAL REVIEW PROCEDURES 50
3.5 SELF/PEER REVIEWS 51
3.6 LAZY DEVELOPER 53
3.7 REQUIREMENTS..53
3.8 SUMMARY 57

CHAPTER 4 – DESIGN 59
4.1 INTRODUCTION 59
4.2 EXTENDING THE IDE 59
4.3 DIGITAL INK 61
4.4 REFLOWING DIGITAL INK 61
4.4.1 Grouping ink strokes 62
4.4.2 Anchoring groups 63
4.4.3 Moving Groups 63
4.5 MODIFYING INK ANNOTATION..64
4.6 ISSUE SEVERITY..65
4.7 PRESERVING INK ANNOTATION 65

CHAPTER 5 – IMPLEMENTATION 67
5.1 INTRODUCTION 67
5.2 OVERVIEW 69
5.3 INK TOOLBAR 70
5.4 INK WINDOW EXTENSION 72
5.5 LINKER 73
5.6 INK GROUPING 76
5.7 INK REFLOW 78

5.8 INK EDITING 80
5.8.1 Moving 80
5.8.2 Erasing 80
5.8.3 Ink properties 81
5.9 ISSUE SEVERITY 81
5.10 SYNCHRONIZING WITH THE INK WINDOW 83
5.11 PRESERVING INK ANNOTATIONS 83
5.11.1 Loading 84
5.11.2 Saving 84
5.11.3 Maintaining consistency 85
5.12 SUMMARY 86

CHAPTER 6 – EVALUATION 87
6.1 INTRODUCTION 87
6.2 COGNITIVE DIMENSIONS ANALYSIS 87
6.3 POST TASK WALKTHROUGH EVALUATION 99
6.3.1 Industry technical review evaluation 100
6.3.2 Marking evaluation 102
6.3.3 Teaching evaluation 103
6.4 EVALUATION RESULTS 105
6.4.1 Architectural extensions 105
6.4.2 Usability difficulties 108
6.4.3 Scenario specific extensions 108
6.5 SUMMARY 109

CHAPTER 7 – DISCUSSION 111
7.1 INTRODUCTION 111
7.2 TOOL DISCUSSION 111
7.3 IMPLEMENTATION DISCUSSION 113
7.4 EVALUATION DISCUSSION 115
7.5 SUMMARY 116

CHAPTER 8 – CONCLUSION 119
8.1 INTRODUCTION 119
8.2 CONTRIBUTIONS TO THIS THESIS 119
8.3 FUTURE WORK 120
8.4 FINAL CONCLUSIONS 124

BIBLIOGRAPHY....................125

LIST OF FIGURES

FIGURE 2.1: Tablet PC 13

FIGURE 2.2: Annotation over Microsoft Products 14

FIGURE 2.3: Testing Procedures 21

FIGURE 2.4: Inspection Feedback 26

FIGURE 2.5: XLibris User interface 33

FIGURE 2.6: Digital Ink Annotation over Internet Explorer 36

FIGURE 2.7: MarkTool User Interface 38

FIGURE 2.8: Penmarked User Interface 39

FIGURE 2.9: FEAT User Interface in the Eclipse Platform 41

FIGURE 2.10: Sketching User Interface Designs in FreeForm 43

FIGURE 2.11: Blog Reader Explorer Interface in Visual Studio 2005 44

FIGURE 3.1: Individual Requirements for each Scenario 56

FIGURE 5.1: An Annotation Example 68

FIGURE 5.2: The RCA User Interface 70

FIGURE 5.3: Ink Toolbar 71

FIGURE 5.4: Line Linker 74

FIGURE 5. 5: Recognized Circle Linker 75

FIGURE 5.6: Annotation Encapsulated within a Red Bounding Box 76

FIGURE 5.7: Annotation within an Extended Bounding Box 78

FIGURE 5.8: Deleting Text Ranges 79

FIGURE 5.9:Indicating Severities 82

FIGURE 6.1: Incorrect Attachment of a Circle Linker 91

Chapter 1

Introduction

1.1 General Introduction

Annotating over printed documents with a pen has been an intuitive way to record feedback. Over the last five years digital ink has emerged as a popular approach for providing feedback over electronic documents. Existing digital ink research suggests people enjoy the added functionality that a digital ink environment provides (Mock, 2004; Moran et al., 1997; Plimmer & Mason, 2006). As a result ink annotation has been an active area of research. Most of the existing research revolves around annotating digital textual documents (Brush et al., 2001; Golovchinsky & Denoue, 2002; Marshall, 1997; Schilit et al., 1998). However there has been little research into supporting annotation of source code. We are interested in determining whether the success demonstrated when annotating digital text documents can be extended to annotation of programming code directly within an Integrated Development Environment (IDE).

This chapter presents the motivation for this research, the goal of this thesis, and our approach. Finally we will give an overall structure of this thesis, briefly describing what each chapter entails.

1.2 Our Motivation

The motivation for this topic explores the user and technical requirements for ink annotation in the IDE and is based on two areas.

Firstly, code review is frequently performed in a variety of situations. For example, as more and more organizations rely on customized developed software it has become increasingly important for developers to write high quality code. Performing a good code review has been proven as one of the best approaches for significantly improving software quality (Colen, 2001; Fagan, 1976). However the current approach of raising issues during a code review includes several time consuming drawbacks, resulting in reviews being poorly performed and sometimes even totally neglected.

The initial downside revolves around the fact that when annotating over printed out documents, the non-linear form of code makes it impossible to read like a book which ultimately leads to a more time consuming review process. Furthermore the non-linearity problem makes it difficult for users as they must manually search through pages of code to find method declarations and classes, as well as locations where specific variables were used.

Another disadvantage is simply writing issues on a separate note-taking device (e.g. paper or text editor) as it causes undue stress on the developer and reviewer. This is due to the fact that they have to switch between the code and the list of issues with no direct point of reference to the code, so the exact location and type of this issue must be described in full detail. The last but equally frustrating time consuming problem occurs when text comments are added to code. These text comments may be coloured differently, however they are represented in the same font-style and font-size as the code, and can be difficult to spot as they look similar to actual code and may be confused with code comments. Using digital ink strokes creates easily distinguishable comments that standout from the underlying code. Ink is an effective intuitive approach when supplying feedback over code. Giving the user a computerized approach to the task adds the customary benefits of computer support.

Secondly, there have been many tools allowing ink annotation over textual documents, though we're interested in whether this success can be extended towards supporting ink annotation over program code. There are a number of tools that allow readers to attach distinguishable ink annotation feedback over textual documents (Janssen, 2005; Schilit et al., 1998), as well as ink annotation support within many commercial products (Microsoft Office Online, 2006). People frequently review code, however currently there is no successful tool that allows a user to attach digital ink directly over dynamically changing source code within a code editor. This makes life difficult for reviewers who want to give feedback over source code.

This thesis investigates whether incorporating digital ink within an IDE will improve the efficiency of code review and will also encourage organizations to effectively perform code reviews. Supporting this functionality within the IDE we also retain all benefits that the underlying code editor has to offer (e.g. object-definition searches, font properties, debugging and execution of code, as well as existing wizards).

1.3 Our Goal

Our main goal is to investigate the requirements of a digital ink annotation tool within the IDE. Such that a code reviewer can bring up a code file or an entire code project and have the ability to generate code issues via digital ink strokes alongside portions of code. Consequently the tool will provide potential benefits to anyone who reads or writes code. The sub goal for this prototype is to explore and evaluate these ideas. As for the secondary goals, we plan to investigate several ink annotation ideas and code review concepts, such that we can include additional functionality in our prototype, above and beyond simple code annotation.

1.4 Our Approach

We approach this project by first undertaking an extensive literature review. This review comprises of a detailed investigation of our two research areas, the first being digital ink annotation, the second being code review. We explore several interesting ink annotation ideas and code review concepts, and then the technical challenges into achieving successful ink annotation are presented. Lastly we explore several tools developed for reviewing different types of documents and tools that extended existing IDE's. Overall this literature review provides a better understanding of what our tool will need to support as well as the issues we will need to overcome in order to satisfy our main goal of exploring ink annotation for code review.

Several code review scenarios and how they are performed are described, we then present how an IDE ink annotation tool could be used as an alternative approach for these scenarios. From our related work and review scenarios we will generate an exhaustive list of features that will benefit an annotation tool, which are then simplified into a smaller set of core requirements. These core requirements satisfy the basic needs for all code reviewers. We then discuss these requirements and possible design implications in more detail. These requirements are then used as a basis for our initial prototype, 'Rich Code Annotation' (RCA).

We scrutinized our tool with two in-depth evaluations, one from the developer's perspective the other from a user perspective. The user evaluation provides feedback on usability, and additional scenario-specific ideas that can be incorporated in future versions.

1.5 THESIS OVERVIEW

This thesis examines the construction, implementation and benefits of an ink annotation code review tool within an IDE. A number of different ideas and implications relating to ink annotation and code review are presented along with our implementation and evaluation of the RCA tool, followed by a discussion and conclusion of this research.

In this thesis we use the term 'issue' to refer to a point of interest within source code. Hereinafter we use the term 'code reviewer' to refer to anyone who reads or writes code, this is discussed further in Chapter 3. We have concentrated on five types of code reviewing scenarios, a teacher/student environment, a marker/student environment, industry development teams, developers conducting self or peer-reviews and finally for a developer who comments over code. This thesis is organized as follows:

Chapter 1: Introduction – introduces this research topic and provides the motivation for this thesis, the goals we want to achieve and our approach to completing this task, and finally the overview of this thesis.

Chapter 2: Related Work – describes both ink annotation and code review as a concept. Following this is a description of related review tools that incorporates annotation over digital documents, and existing tools that have been integrated within existing programming environments.

Chapter 3: Scenario of Use – investigates five separate code reviewing scenarios, and the advantages and possible features this tool could provide for each scenario.

Chapter 4: Design – presents a description of core requirements needed for an ink annotation tool that resides within the IDE.

Chapter 5: Implementation – describes our implementation and the issues we faced while creating this ink annotation code review tool.

Chapter 6: Evaluation – evaluates our ink annotation tool by investigating the strengths and weaknesses through both the developers and the user's perspective.

Chapter 7: Discussion – presents an overview of the tool, its implementation, and its evaluation that includes our experiences, difficulties and possible improvements to the base implementation.

Chapter 8: Conclusion – summarizes the contributions this research has on ink annotation, code review and IDE integration, followed by future work consisting of scenario-specific improvements and further ink annotation research.

1.6 Summary

This chapter introduced our thesis topic and the two areas of research that this project is centred around. We presented the motivation for this research along with our overall and secondary goals and finally outlined our planned approach towards this task. We now begin this project by first presenting concepts and related research, and then our subsequent chapters will present the work for our thesis as outlined in the thesis overview.

Chapter 2

Related Work

2.1 INTRODUCTION

This chapter outlines the related work for this thesis. As this research project investigates the integration of digital ink into a coding IDE, such that reviewers can then supply feedback directly over code documents. We first provide a deeper understanding of this project by introducing some necessary background concepts along with its benefits, challenges and difficulties. A variety of related tools are presented, these provide some valuable ideas that we can incorporate into our ink annotation tool. Along with these related tools, we also present several tools that have been successfully integrated into existing code IDE's and the benefits they now provide to their users.

2.2 INK ANNOTATION

Ink annotation has been used over both paper documents and digital documents to highlight key ideas, supply feedback and to indicate information for future recall. There exists a variety of different ink annotation research areas, ranging from types of annotations, their functions and their popularity as well as issues in supporting ink annotation over digital documents.

This section discusses existing research in traditional ink annotation as well as its advantages and disadvantages. A further description to why an effective approach is needed to support annotation directly over digital versions of documents. Then the contribution that digital ink has on annotating documents is discussed, along with its advantages in a typical annotation environment and the more extensive advantages specific to educational and collaborative environments. Finally some technical challenges related to digital ink that an annotation tool must overcome to ensure users don't revert back to traditional pen and paper methods.

2.2.1 TRADITIONAL INK ANNOTATION

Reading is a highly practiced task and performed for a variety of different reasons (Lorch Jr. et al., 1993; Oakhill & Garnham, 1998; O'Hara, 1996). For example, text can be read for enjoyment, general knowledge, or for a particular task e.g. studying, marking, proof-reading (Shipman et al., 2003). Independent of the reader's motive, many choose to annotate over their paper version with a pen, this is considered a traditional pen and paper annotation approach. Annotation over a document is the readers approach to better digesting the content, identify important passages for future recall, or even to communicate their interpretation with future readers in a shared environment. This type of reading with critical thinking and learning is considered 'active reading' (Adler, 1972).

There are three main styles an annotation can be classified as, these are explained in Marshall (1997) and include: annotations within text, annotations and notes in margins or blank space, and removable annotations. Annotations within text include brief notes between lines as well as the less-interpretable annotations such as underlining, highlighting, and circled words and phrases. Annotations in margins or in blank spaces range from margin comments that describe a portion of text, brackets and braces that explicitly select a range of text, and symbols (asterisks and explanation marks) where only the reader could understand its context. Finally, texts may even include removable annotations that reside between pages, including notes on separate paper or post-it notes stuck on pages. As margin comments involve extended notes with a clearer elaborative meaning; these are the types of annotations that would be important when dealing with code annotation. Code documents generally contain a large amount of blank space that margin comments could easily occupy. Also each annotation should have a clear elaborative meaning as these issues are written by a reviewer and passed on to the developer to read and resolve, so again margin comments would be most appropriate.

Marshall (1997) also investigated the functions behind annotations within an educational setting; these exclude the unrelated doodling and drawing annotations (for example "I LOVE YOU"). First, underlining and highlighting of headings and sections signal areas needing future attention or re-reading. Second, highlighting, underlining and circles that encapsulate a word or several words are used as a way of remembering an idea for future recall. Third, in-depth descriptive comments, figures or equations written in the margin (marginalia) indicate an analysis of the content or the solution to a problem near the content. Active readers write extended notes in the margin of the source document rather than risk a break in ones attention span (Marshall, 1997; O'Hara & Sellen, 1997). Forth, short notes in the margin are generally used to record ones interpretation or narrative feedback of a nearby sentence or paragraph.

Finally, the annotation most readers are accustomed to is the extensive use (pages and pages) of highlighting or underlining (Bargeron & Moscovich, 2003; Shipman et al., 2003), which is generally used as a visible trace for keeping the reader's attention through difficult passages of text.

Previous research in types of annotation used over documents indicates the majority of annotations revolve around highlighted and underlined passages (Bargeron & Moscovich, 2003; Marshall & Brush, 2002; Shipman et al., 2003). In fact the study in Marshall (2003) found over 82% of the 1500-plus annotations consisted solely of underlines, highlights and circles. It was found in Bargeron & Moscovich (2003) that 31% of annotations were underlines and 27% of annotations were highlighting, followed by marginalia annotations at 24%. Both circling passages and margin bars were the least common at 12% and 6% respectively. These results are also consistent with the statement made in by O'Hara & Sellen (1997) where most personal annotations are simply anchors (underlining, highlighting) to a portion of text without any comment content. For an annotation tool it then seems support for underlining, highlighting and marginalia would be important. However, Marshall (1997) mentioned highlighting and underlining was used to signal sections for re-reading and indicating words needed for future recall, and finally as an approach to keep a readers attention. A code review tool is simply used to indicate coding issues and perhaps supply feedback to the developer, it would be more appropriate to incorporate marginalia comments rather than highlighting and underlining.

During a documents lifetime it is common for personal annotations to be read by others either intentionally or unintentionally (Marshall, 1997). This is essentially how code review annotations should be treated. Marshall & Brush (2004) investigates whether personal (private) annotations are suitable to be shared with other readers. It was agreed that only a small portion of personal annotations could be shared and understood by others (Marshall & Brush, 2002; Marshall & Brush, 2004). It was found that annotations only containing a text anchor point and no content would not be appropriate for a shared environment. Whereas, annotations that support a comment along with a point of reference (anchor + comment), supply better communicative intent (Marshall & Brush, 2004). Having clear, understandable annotations is also favoured by students buying 2nd hand textbooks, where students prefer textbooks with easily interpretable writing in the margins and without pages and pages of highlighting (Marshall, 1997).

So when developing a code annotation tool where issues are generated by a reviewer then passed to the developer to resolve. It is extremely important that the developer can understand the annotations made by the reviewer, hence no ambiguities with the given feedback. Therefore the annotation tool should entice the reviewer to attach a comment along with the portion of code that relates to the issue so it is easily understood. The anchor could simply be a circle, a margin bar or a line to indicate the point of reference, and the description could reside in the blank spaces of the code.

Traditional pen and paper annotation has many benefits that make it more favourable than using a personal computer, there are also many disadvantages of annotation with paper and pen that can be overcome by computers. Bargeron & Moscovich (2003), Brush et al. (2001), Huang et al. (2003), Marshall (1997), Marshall (2003) and Schilit et al. (1998) elaborate on the advantages and disadvantages with traditional ink annotation, a summary of these are presented below:

Advantages of Traditional Pen and Paper

Availability

Paper, pens and other writing devices are inexpensive and easily obtainable, and most books can be commented on. When digital documents need annotating, they can be simply printed out with little effort and cost.

Free-form

Any types of markings can be made absolutely anywhere on the page, with no limitations or constraints.

Mobility

Pen and paper annotation is very portable and convenient. Documents can be annotated almost anywhere for instance while relaxing in bed, on the couch or even on a boat, train or plane. In fact 90% of people who participate in active reading say they would rather a printout of the document such that they can read in a different setting (Marshall, 2003).

Multiple Viewing

Having loose bits of paper allows side-by-side representation making it easier to transfer ideas from one page to the other. It can be common for readers to have many books open on their desk while they write on another.

Noticeable

Annotations are distinguishable from the underlying paper document as they are made in a different font style, size and generally a different colour.

Social

Interacting with a paper document whether it be reading, writing or drawing, can be performed more effectively by a small group of two to three people, than when interacting on a desktop PC. Social interaction like this can even, in some cases, be done simultaneously (depending on the size and context of the document).

Spatial

The area of paper that can be annotated is not limited by the size of the page, for more elaborate and larger comments extra paper can be stapled, taped or left between the pages. Some books contain blank pages where notes/formulas, even post-its are easily added to a page.

Tangible

Paper documents are physical artifacts, they can be folded, rotated and cut in half, papers can be easily reorganized and the size of a document and its pages are easily visible. Physically seeing the document on your workspace also serves as a tangible reminder that it needs attention (Marshall, 2003).

Disadvantages of Traditional Pen and Paper

Incomprehensible

Personal annotations may make sense to the previous owner of the document but not to the current reader, after a few weeks the owner may also lose the meaning of their annotations. Annotations may be written incorrectly, giving the reader a false understanding of the situation. Annotations may have been written in such a rush that the reader doesn't understand what has been written.

Pen Problems

Some forms of pens and highlights can 'bleed' on a page leaving an unsightly mess on the page, the ink may even be strong enough to be absorbed through to the next few pages. Pens and highlighters unfortunately dry up, run-out of ink or get misplaced, and some pens only allow the annotator to write flat on a desk rather than an upright whiteboard position.

Reproduction

Reproducing multiple copies of a document using a photocopier or printer can be difficult and time consuming to find and even operate, some only print in greyscale. It would be much more convenient if the document didn't need to printed-out in order to annotate it.

Storage

Having many documents and loose pages stored on your work desk can build up, creating an untidy working environment. Documents that are un-organized may accidentally get thrown out, or lost among other documents. Storing documents in filing cabinets may also be difficult to locate in the future.

Transferring

Once feedback is annotated over a document it may need to be hand delivered or mailed to the colleague. Because paper is tangible you have to physically get up and perhaps leave the office, get in the car to pick up the annotated document from home or even deliver it to someone else.

Unchangeable

When using a pen over a textbook the annotations will forever remain and can only be concealed with the use messy ink remover or by printing-out a new version. Annotations on textbooks are final, a new textbook in pristine condition may be difficult and costly to obtain.

2.2.2 Digital Document Annotation

In this electronic era "much of what we read when we research a topic now arrives in digital form" (Marshall, 2003). Five years ago traditional pen and paper was thought of as being the only practical way to comment on a document (Crawford, 1998; Dillon, 1992). This is also supported with the results in O'Hara & Sellen (1997), when choosing to read a document "the benefits of paper far outweigh those of on-line tools" (O'Hara & Sellen, 1997).

When annotating over digital documents the advantages of pen and paper make it a preferred medium for active reading and hence encourages the reader to read a printed-out version (O'Hara & Sellen, 1997; Schilit et al., 1998). This in-turn produces a large amount of paper, which simply gets thrown out when finished with, in fact Johnson (1997) predicted that 1,344 billion pages will be generated by printers and photocopiers in one year alone.

It's clear, that the ability to comment directly over digital documents incorporates a computerized approach, which results in a number of the disadvantages of traditional annotation being eliminated. Comments could be plainly editable and a different variety of colours and fonts could be quickly selected without the chance of running out of ink, and if annotations ruin the state of the document then "there would always be a pristine version on the floppy disk" (Stein, 1995). Using the computers file-system the document and corresponding notes can be easily stored, searched, and transferred via the internet. However for this computerized annotation to become effective and adopted by annotators, it is clear both new hardware and software is required that supports a convenient annotation practice directly over digital documents (O'Hara & Sellen, 1997).

2.2.3 DIGITAL INK ANNOTATION

In 2002, tablet PC's were released with the aim to provide PC users a practical and natural approach for annotating their digital documents via digital ink. The tablet PC (shown in figure 2.1) is essentially a laptop computer running on a Windows XP Tablet operating system, and allows the screen to be lifted in an upright position, rotated 180 degrees then folded back down over the keyboard (Microsoft Windows XP Tablet PC Edition, 2005). Equipped with a stylus, the user supplies high precision digital ink input directly over the single input tablet screen. With its lightweight portability, tablets have become the most intuitive, effortless and convenient approach for supplying feedback and flagging key ideas over digital documents. There also exists an 'Ink Annotation software development package' (Microsoft Download Centre, 2006), allowing developers to create their own software supporting digital ink strokes over their user interface.

Figure 2.1: Tablet PC

Digital ink has been used for several different tools that allow free-form annotation strokes including highlighting, underlining and the drawing diagrams directly over digital documents. For instance, several Microsoft Office products (Microsoft Office Online, 2006) support digital ink overlapping their documents (shown in figure 2.2), XLibris (Schilit et al., 2003) and ReadUp (Janssen, 2005) also supports loading and storing digital ink over text documents. Ink annotation has also been used successfully to supply ink feedback over web-documents (iMarkup Solutions, 2006; Ramachandran & Kashi, 2003), and student assignments (Plimmer & Mason, 2006), these are described later in this section.

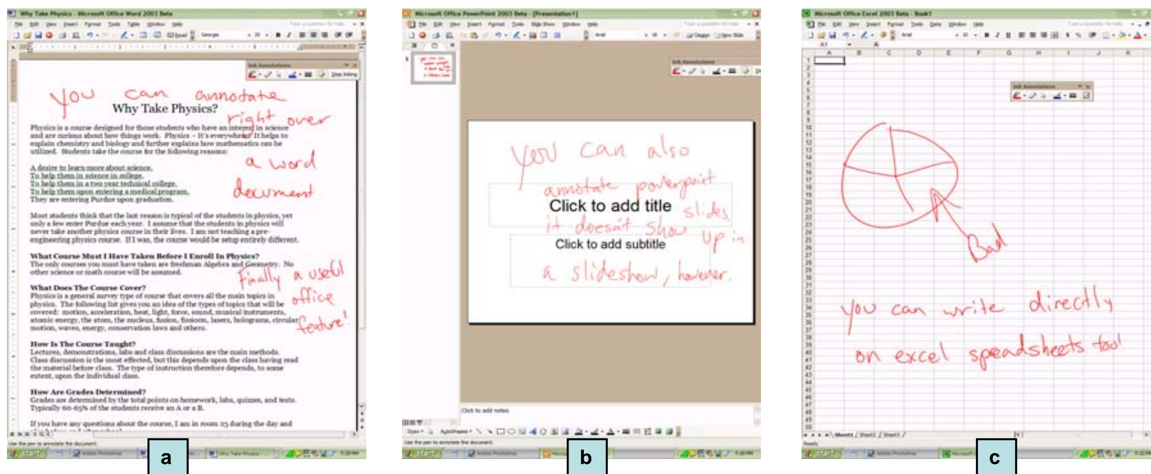


Figure 2.2: Annotation over Microsoft Products includes: (a) Microsoft Word, (b) Microsoft PowerPoint, (c) Microsoft Excel

Ink annotation over digital documents eliminates most of the disadvantages evident in the traditional pen and paper approach. Our digitally annotated document can now be easily stored and located on the hard-drive, it can be effortlessly transferred via internet, and finally there is no need for printing. Presented below is a summarized list from Huang et al. (2003), Marshall (1997), Marshall (2003) and Plimmer & Mason (2006) of further advantages that digital ink offers:

Advantages of Digital Ink Annotation

Intuitive

Annotating with digital ink is a similar approach to traditional pen and paper annotation. It's a natural process that is familiar to most, and annotations can be made just as easily and as neatly with digital ink as if we were using traditional pen and paper.

Clean Editing

As the ink strokes are digital, they can be easily added, deleted, moved, copied and pasted, annotations can also be selected and modified as a group. No messy erasing is needed, and if the document contains too many annotations a clean copy can always be obtained.

Digital

Extra information can be stored along with each annotation, including the author, the time created or its severity, which can be used for future reference. The ink strokes are also digital, meaning they can be searched and filtered based on its location on the page, its severity, or its ink characteristics (ink colour, time written, and type of stroke).

Distinguishable

Much like traditional annotation, digital ink stands-out from the underlying document. It involves human strokes that are distinctly different from the documents underlying font-size and font-style. This also makes it an effective approach when skimming the document for feedback notes, because a glimpse of digital ink is easily spotted when quickly scrolling through the document.

Existing PC Functionality

As the user annotates a document outside the office, they also have applications to support their annotations. For example as one annotates over text documents, one also has access to dictionaries and thesaurus. When the documented material involves formulas and numbers, users have access to the on-screen calculator to verify sums. When annotating research or documents involving web links, then the user can easily annotate the document while simultaneously browsing the internet.

Variety of Pen Instruments

A digital ink tool can include a range of pen types with different thickness as well as a variety of colours all at the tap of a button. The different annotation instruments include: ball-point, felt, calligraphy, highlighters, and underlining pens that only work in a straight line, different size erasers can also be included.

Tablets have been met with praise by educators (University of Notre Dame, 2005; Wachsmuth, 2003) and further studies in this field also show teaching with a tablet PC increases students attention and understanding during class (Anderson et al., 2004). Mock (2004) investigated the effectiveness of tablet PC's in a teaching and learning environment, the study produced positive feedback from students where they preferred this approach to traditional blackboard style teaching. Mock (2004) conducted this study by replacing the blackboard with a tablet PC that was connected to a data projector which was viewed by students in the classroom. Rather than annotating over a blackboard or whiteboard, the class notes and the in-class annotations were now presented over the tablet PC and viewable by all. Therefore Tablet PCs work well in an educational environment, and so presents an opportunity for a digital ink coding tool to be extended for use in a teaching environment. Digital ink annotations can then be made directly over the code editor to describe coded examples. Along with the general ink annotation advantages above, Mock (2004) discussed several teaching and learning benefits, and are presented below:

Advantages of Classroom Digital Ink Annotation

Clean Annotation Display

Annotations are made directly over the tablet display and projected onto the board, rather than annotating directly on the board with your arm elevated in the air. Thus giving benefit to annotators, it also involves no messy chalk dust, no nasty ink fumes from the whiteboard ink pens, and no need to clean the board before and after class.

Clear Viewing

A tablet PC can be connected to a projector either with a cord or wirelessly, this breakthrough takes away the need for the lecturer to stand in front of the board. Which presents obvious benefits to the students as the annotator doesn't obscure the annotations as they are reading or writing them. The lecturer can also walk around the room and annotate or have student's annotate without having to be at the front of class.

In-Class Annotation

With digital ink a lecture can be conducted without the need for preparing material, or alternatively a limited amount of material can be prepared which can be elaborated upon with digital ink notes during the lecture. For this reason the lecturer will gain a natural annotation mindset when teaching, for instance the teacher could discuss coded-examples line-by-line with written annotations describing the purpose for each code line.

Reproduce Previous Annotations

As annotations are digitally written over the lecture material, they can be stored with the document and put-up on the course web-site for students to read in the future. This benefits students as they can concentrate on listening to presented content rather than spend the entire lecture vigorously writing down notes. This also benefits the lecturer as notes can be easily reproduced during class when an elaboration is needed from a previous lecture, where traditionally these notes would have been lost.

Tablet PC's can be connected wirelessly to local networks allowing users to publicly share their own personal annotations to others in real-time. This collaboration presents an effective means of communicating ideas and delegating tasks with digital ink when working on the same document. This can be related to annotating over the same code file within a team development environment or when sharing annotations over the same lecture material. Collaboration of notes is especially important in industry development environments, as developing and reviewing code is often done in teams, this will be discussed later in this section.

Sharing of notes over the same education material also presents a learning advantage for students, resulting in a greater learning experience. Simon et al. (2004) performed a collaboration study where each student was equipped with a tablet PC during a class. Students would make notes during class which are all submitted to the instructor, who can then choose what annotations to display on the projector. Huang et al. (2003) investigated a similar collaboration concept where each student equipped with a tablet PC could publicly share their annotations with others, students found the material easier to understand as they were explained from other students perspective.

2.2.4 Technical Challenges

As valuable as digital ink may sound, there are some challenging difficulties that must be addressed to ensure users will continue to use this approach without reverting to the traditional way of annotating. The typical challenges apparent when providing ink annotation include: tangibility factors and document navigation that exist with traditional annotation (Schilit et al., 1998); the ability to recognize (Rubine, 1991) and beautify digital ink (Plimmer & Grundy, 2005); providing separate ink layers over the document where ink must remain aligned with its underlying text (Golovchinsky & Denoue, 2002; Plimmer & Mason, 2006); grouping and moving ink to support changes in the underlying text (Bargeron & Moscovich, 2003; Brush et al., 2001; Golovchinsky & Denoue, 2002).

In some annotation tools, beautification support may be encouraged as it provides a legible and professional representation of an annotators ink strokes. This is especially useful when annotating during a class or a code review meeting, as time limitations may cause untidy handwriting. Beautification may benefit a code review tool however it is outside the current scope of this initial proto-type. As this thesis is more interested in supporting digital ink annotation over editable code within an IDE, the main challenges encountered involve: separate ink layers to support digital ink over code; and handling ink strokes as the underlying text/code changes (reflow). These are discussed below:

2.2.4.1 Seperate Ink Layers

Using a separate transparent layer above the underlying document is a common approach to support digital ink strokes in an annotation tool. For example, ink strokes are made on the transparent layer, and as the underlying tool scrolls up/down, so would the transparent layer, thus giving the illusion that the ink is

directly over the document. Controlling this layer can be difficult, as the underlying document window may resize, scroll, or be viewed with a different aspect ratio or a different font (Golovchinsky & Denoue, 2002; Plimmer & Mason, 2006). These reasons mean the transparent ink window may need to adjust its own size and the positions of its ink strokes to remain consistently aligned with its underlying context.

Schilit et al. (1998) provides digital ink annotation over PDF documents and takes a simpler approach. It dedicates the entire display to one page, and rather than scrolling to the next page, the tool uses next-page and previous-page buttons, so annotations remains fixed on the page and there is no need for document scrolling.

Ramachandran & Kashi (2003) investigates ink annotation in a web browser by using a transparent 'activeX' component that captures ink strokes, it then uses scrolling offsets and mouse coordinates to attach the ink strokes to specific web elements. The document object model (DOM) and dynamic HTML (DHTML) functionality of the browser provides support for capturing events, which is used to reposition the ink annotations.

Plimmer & Mason (2006) developed a code marking tool that also uses a separate transparent ink layer that remains over the front of a code document. This tool uses scroll bars over the transparent layer not the underlying window, otherwise scrolling will bring the underlying window above the ink annotation layer. So as the user scrolls the transparent layer up or down, the underlying document moves up or down to give the illusion that the scrollbar and the digital ink is directly attached over the document.

2.2.4.2 Reflowing Digital Ink

Annotations over paper documents are easy to handle as its textual content is fixed, however when annotating over a digital text documents its content is generally editable. Once someone has marked-up (annotated) the document with digital ink, the textual changes can be difficult to handle. Words or lines can be added, deleted or even moved to a new location, meaning the annotations must also follow. As the content changes, the digital ink loses the link to its attached content, resulting in the annotation pointing to the wrong text. Digital ink must automatically reflow (reposition) itself either vertically, horizontally, or both to remain consistent with its underlying text.

Existing ink reflow research revolves around three major steps and is described further in both Bargeron & Moscovich (2003) and Golovchinsky & Denoue (2002). First, every ink stroke must be added to a specific group, ink can be part of a new annotation group or placed in an existing group of nearby annotations. When deciding what group a newly created ink stroke should belong to its common to take into account both the time and location of strokes. The time between the previous ink stroke and the current stroke is considered a temporal relationship, and the location of the current stroke to nearby strokes is considered a spatial relationship. For instance, if the letter 'i' was made a very short time after the letter 'F', then chances are they correspond to the same word ("Fix"), so based on a temporal relationship the strokes are grouped together. However if the annotator forgot to dot the 'i' and went back to this minutes later, then the dot would be sufficiently close to be considered part of the same annotation group (spatial relationship). Golovchinsky & Denoue (2002) investigated how strokes should be grouped. The results showed the average time between each stroke is approximate 250 milli-seconds. The average time between the completion of the last stroke of a word and the start of the next stroke of a new word is approximately 500 milli-seconds. This research concluded, two strokes that occur within 500 milli-seconds of one another should be grouped together.

The second step involves anchoring (attaching) an annotation group to a specific word, sentence, paragraph or figure within the document. This step can be challenging as it's difficult to decide the exact piece of text that the user wants the annotation to be attached to. Another problem with attaching ink to text anchors is reflowing ink when the anchored text is modified, moved or deleted. It can be difficult to know the new location of the annotation when there is uncertainty in finding the correct anchor (Bargeron & Moscovich, 2003). However this problem can be reduced by using both a specific text anchor and also considering the anchors surrounding text (Bargeron & Moscovich, 2003) or even orphaning the annotation at the bottom of the page if its location can not be decided (Brush et al., 2001). So, as the anchor is modified the annotation can still be positioned by locating similar surrounding context, if there is any uncertainty a message could be sent to the user to decide the outcome of the annotation.

Finally, when text is added, deleted or moved it causes existing text to reposition itself, the ink must also move in the same manner as the text. The challenges seen when re-rendering these annotations occur when the document is modified, as some annotations may need to be physically reshaped. These could potentially look visually different to the previous annotation and may even be disorientating to users and not meet their expectations (Bargeron & Moscovich, 2003). This problem is particularly visible as underlined or circled annotations must be split over line breaks and page breaks. However, this is generally only seen in text documents, as coded documents only involve one code sentence per line, meaning annotations won't be attached an entire line and part of the line underneath.

2.3 CODE REVIEW

“Software technical reviews are essential to the development and maintenance of high quality software” (Collofello, 1987). When performing any type of code review independent of any specific review scenario it is important to have an effective approach for recording reviewer’s comments. This is the basis for our research and is discussed further throughout this thesis. The software engineering process is defined as an organization’s technical and managerial activities that transform a user’s requirements into software (Joh & Mosley, 1989). However the process of developing software deliverables doesn’t just imply developing a program that simply works, once the software is developed it must be tested.

There exist many testing approaches available for adoption by software development organizations, these are presented in Chillarege (1999), Chen (2005) and McGregor (2006), and are summarized in figure 2.3. These testing procedures all improve the quality of development projects. However ‘Code Review’ is a technique that stands out from the rest, and should be performed throughout the development, not just once the project is completed like other methods. Some say code review is “grown to be recognized as one of the most efficient methods of debugging code” (Chillarege, 1999). ‘IEEE Computer Society - Software’ promote it as being one of the best influences seen in the last 50 years of software engineering (McConnel, 2000). Code reviews are performed throughout the development, where code is brought up on a computer screen, overhead projector, or is printed-out and distributed to the review participants. The code is described and then issues are raised by the review participants, which are then resolved by developer before any further implementation can begin.

An ink annotation tool within an IDE is essential for indicating points of interest within code, and can then be incorporated into an industries code review process. The first type of code review approach was a code ‘walkthrough’, and was a common review approach for organizations such like IBM (Ciolkowski et al., 2002). Fagan (1976) then presented an inspection procedure that took the walkthrough approach and applied formal stages and roles, and still stands as a well recognized inspection methodology.

Both industry review approaches have their advantages. They are recognized as a cost effective technique of finding errors early within the development lifecycle and result in an increased chance of issues being resolved properly (Colen, 2001). Reviews also result in others having a good understanding of the project, who can then continue developing the project if the developer becomes unavailable (Colen, 2001).

Test Procedure	Description
Entry and Exit	Module/Procedure/Component is tested, ensuring the specific input generates the correct output.
Functional	Similar to above, however the entire function is checked to ensure correct output for a particular command/option, also know as black-box testing.
Component	Tests individual classes or tests several tightly coupled classes together.
Automated	A program is written to automatically run a number of test cases.
System	Tests the efficiency in terms of processing time under normal conditions (known as performance testing), also under stressful conditions where the system is pushed beyond expected requirements (stress testing).
Usability	Tests to determine how well people can use the developed artefact.

Figure 2.3: Testing Procedures

We are now going to present a brief description of both forms of code review (walkthrough and inspection), as well as some interesting facts that resulted from performing these types of reviews. The formal code inspection methodology was first introduced by Fagan (1976), this was the basis for further research in code review procedures (Johnson, 1994; Knight & Myers, 1991; Parnas & Weiss, 1985) and so we provide an in depth summary of Fagan's (1976) paper. It contains a good description of common code review concepts, including when they should be performed, how they are performed as well as some interesting results and benefits of this review process. The differences between walkthroughs and inspections are then compared, as well as determining the best code review approach for a given situation. Finally difficulties that are seen while performing code reviews are presented along with possible solutions to overcome these difficulties.

2.3.1 WALKTHROUGH

A code walkthrough involves the developer explaining portions of code, or individual lines of code to themselves or someone else in an informal way. A walkthrough can be described "as someone presenting the entire logic of an artefact (code) and paraphrasing it aloud, while others listened and asked questions" (Ciolkowski et al., 2002).

Walkthroughs can also be done in a team environment. Karat et al. (1992) did a comparison of the effectiveness of finding problematic pieces of code (issues) when performing a walkthrough individually and also as a team. It was concluded that conducting a walkthrough as a team was more effective in terms of the number of issues raised than when the process was performed individually. Karat et al. (1992) also found that group walkthroughs were more efficient at finding each error (low number of hours per issue found).

Team walkthroughs were found to be more effective at finding issues, however it was also more time consuming, taking an approximate extra 25 hours to complete. So when choosing a team walkthrough, there is an obvious trade-off between having better quality code or reduced labour hours.

2.3.2 INSPECTION

"An inspection is the most systematic and rigorous type of peer review" (Wiegers, 2001).
Formal inspections first emerged through M. E. Fagan in a paper called "Design and Code Inspections to Reduce Errors in Program Development" (Fagan, 1976). Inspections involve a more formal approach to code review, both design and implementation are reviewed, and if done with care can produce much higher quality deliverables. The general approach to this formal inspection revolves around a multi-stage reviewing procedure that typically involves the following stages: planning, overview, inspection meeting and ending with issue resolution. Inspections also involve a group of participants with specific roles similar to that in Fagan (1976), these participants would meet several times throughout the development lifecycle to discuss and inspect the project.

Inspections have been used successfully in many programming projects, including both system and application projects of varying size (small to large), and due to the constant evaluation and group meetings throughout the project's lifecycle, inspections enables higher predictability, improved productivity and product quality (Fagan, 1976; Arthur, 1993; Wiegers, 2001).

Inspections are not only more effective at discovering errors than testing and walkthroughs (Johnson, 1994), others say they are also effective at discovering different classes of errors (Basili et al., 1986; Myers, 1978). With this in mind, at the conclusion of an inspection, and with the help of feedback reports, the developer and his team will be more aware of the number of issues as well as the type of issues raised. This will benefit future inspections for the project, as the team will be more aware of how many errors and the types of errors to lookout for, this will also benefit the developer in knowing what errors he/she is

more prone to make. These categorizations, backup the claim for the need for different ink colours such that issues should be colour-coded as they are recorded to represent different error-types.

An organization can incorporate a 'Trusted software Methodology' as a means of assessing and improving the trustworthiness of either a new or existing project (Amoroso et al., 1994; Daily & Foreman, 1984; Humphrey, 1989). This provides the inspection team with a set of software trustworthiness guidelines and standards to improve any engineering and security problems that may arise in their projects. For instance, one participant can concentrate on locating and raising security problems of the project, while the other uses a checklist of common software engineering problems to reduce the likelihood of errors in the design and implementation.

2.3.3 M. E. Fagan

Code Review as was first formally described in "Design and Code Inspections to Reduce Errors in Program Development" by researcher Mike. E. Fagan (Fagan, 1976), this paper discussed code review in some detail. A formal inspection can be performed after any clean compilation throughout the development of the project, and before the final testing phase of the projects lifecycle. The paper first outlines the necessary approach to conduct a formal design/code inspection, in terms of the inspection stages and the roles of the inspection participants. Finally the paper concludes with a summary of this code review approach and some interesting results.

2.3.3.1 Conducting a Formal Code Inspection

Fagan (1976) states that an inspection team is best served when they have specific roles, this is also consistent with other meeting type scenarios where they use a group of reviewers that includes a moderator or leader to regulate the meeting/review (Collofello, 1987; Johnson, 1994; Nunamaker et al., 1991). Fagan also explains that four people constitute a good sized inspection team and that the team size should not be increased over four. However he then suggests, if the code being reviewed was developed by more than one programmer, then it would be profitable for the other developers to also be involved in the inspection.

The roles of each participant and how the right candidate should be chosen in terms of Fagan (1976) are explained as follows:

Moderator

The moderator is the person who runs the entire inspection; they must recruit and manage a group of reviewers throughout the inspection as well as schedule meeting times. They must offer mediation during group discussions and also produce an inspection review report containing the issues raised during the group inspection meeting (generated at the conclusion of the group meeting). Therefore the moderator would need to be well-respected with good time management and leadership skills.

Scribe

The 'scribe' is someone who records the issues during an inspection meeting and in this case is handled by the moderator. Although research has shown it's beneficial to use someone else rather than the moderator, as it can be an extremely difficult to accurately capture and illustrate the issues/errors while at the same time moderating the inspection (Johnson, 1994).

Designer

The designer is the individual responsible for designing the program (not necessarily the code developer), if this designer also implemented the program code then they are only considered the code designer.

Code Developer

The developer is the one responsible for converting the program design into code. However if this coder also fits in the designers role, then another developer who has preferably worked on a similar project should assume the role of the code developer.

Tester

The tester is accountable for writing and/or executing test cases, and therefore tests both the design and implementation. Again, if the tester was also the code developer, then another tester that preferably has testing experience with a similar project should be assigned the role of the tester.

Once the inspection team is chosen, an inspection can be performed several times throughout the projects development whereby each inspection includes specific stages. A description of each stage within an inspection is given below. Fagan also suggests abiding by a specific time-limit for each stage, and as more projects are developed, the scheduled time-limit for each stage can be adjusted based on historical data of previous project inspections. The inspection process starts with an overview of the project, a preparation of the reviewed material, then a group discussion. This early knowledge from both the overview and preparation gives each reviewer a higher understanding of the project, and better prepares them for future development and inspections within the same project (Fagan, 1976; Johnson, 1994). These stages are discussed below:

Overview

This is the first stage of each inspection and is performed in a group environment with all participants. Documentation of both the design layout and the code implementation is distributed to all inspection/review participants. The designer or coder (depending on inspection type) then describes the overall design/implementation as well as individual aspects of the artefact. The moderator would then highlight the issues and discuss the issues which were resolved as a result of the previous inspection.

Preparation

Once each reviewer has a clear overview of the current project status, they individually prepare for the group inspection stage that follows. This is where each reviewer would study the parts of the project that will be described in the next stage (group meeting). They would attempt to understand the design/implementation intent and logic, such that they can supply both positive and negative feedback during the group review. However there is one disadvantage with this stage, it is possible for the participants to prepare inadequately, and hence attempt to "bluff their way through review process" (Johnson, 1994). Two possible remedies for this were presented in Johnson (1994). The first remedy removes one page of the review material when distributed, and those who prepared for the next stage will notice this absence and contact the review leader (moderator). The second possible remedy is to have participants generate their own private list of issues on paper that they want to present in the inspection stage, this gives the moderator a good indication of each reviewer's preparation.

Inspection

This is the group meeting stage, where the main objective is for the review participants to raise issues and find errors. High-level documentation including both the design and implementation specifications are made available to each review participant, and usually the moderator chooses someone independent to the coder to read portions of code to the group. When presenting the code to the group it is important that each piece of logic and code portion is agreed on and each possible coding execution path is covered.

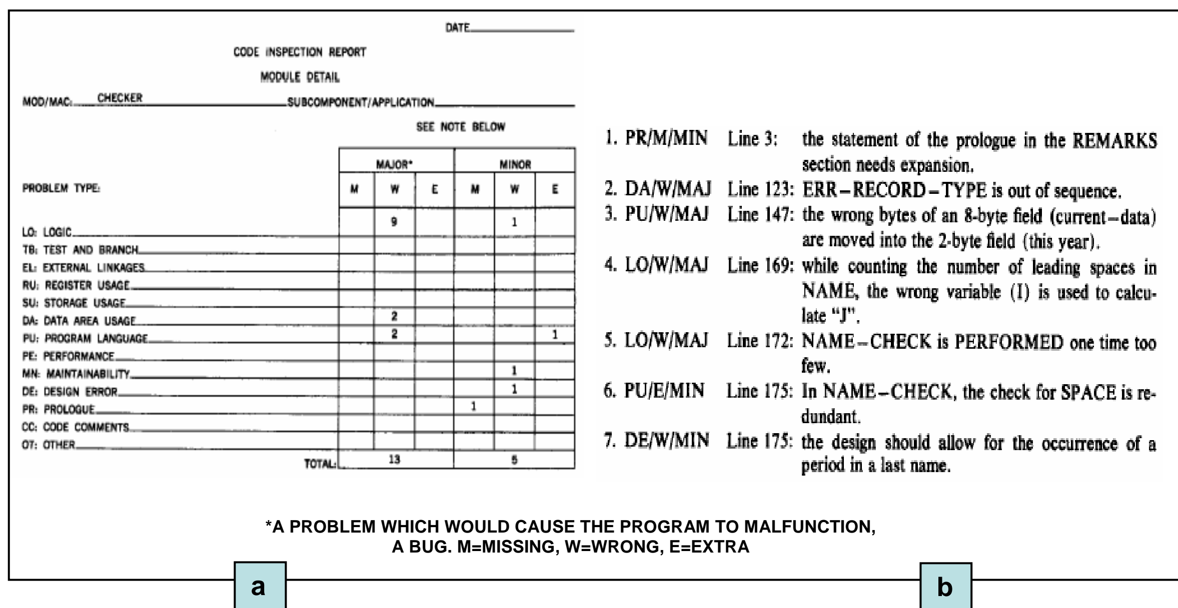
Each issue found must be noted down by the scribe (generally the moderator) along with its line number, a description of the issue, the issue type and an agreed upon severity (either major or minor). Fagan (1976) mentions that it's important these issues are merely noted and not solved as this can lead to further time consuming discussions and debates. Once all the issues are raised and documented, the reader (coder) would propose an approach for the next part of implementation. The final part of this stage occurs once the meeting is over, the moderator generates an 'inspection feedback report' that contains all issues raised in the meeting (shown in figure 2.4), this is given to the developer (or designer, if in the design phase) to resolve.

Rework

This stage requires the designer/developer to resolve all issues listed in the 'inspection feedback form'. These issues must be resolved before any further development takes place.

Follow up

This is the last stage of an inspection, where it is the moderator's responsibility to ensure all issues presented to the designer or the developer are completely resolved. If an issue hasn't been resolved in this stage, then results in Fagan (1976) indicate an extra 10 to 100 times more programming time is required to fix this later in the development lifecycle.



DATE________

CODE INSPECTION REPORT

MODULE DETAIL

MOD/MAC: CHECKER SUBCOMPONENT/APPLICATION________

SEE NOTE BELOW

PROBLEM TYPE:	MAJOR*			MINOR		
	M	W	E	M	W	E
LO: LOGIC		9			1	
TB: TEST AND BRANCH						
EL: EXTERNAL LINKAGES						
RU: REGISTER USAGE						
SU: STORAGE USAGE						
DA: DATA AREA USAGE		2				
PU: PROGRAM LANGUAGE		2				1
PE: PERFORMANCE						
MN: MAINTAINABILITY					1	
DE: DESIGN ERROR					1	
PR: PROLOGUE				1		
CC: CODE COMMENTS						
OT: OTHER						
TOTAL:	13			5		

1. PR/M/MIN Line 3: the statement of the prologue in the REMARKS section needs expansion.
2. DA/W/MAJ Line 123: ERR-RECORD-TYPE is out of sequence.
3. PU/W/MAJ Line 147: the wrong bytes of an 8-byte field (current-data) are moved into the 2-byte field (this year).
4. LO/W/MAJ Line 169: while counting the number of leading spaces in NAME, the wrong variable (I) is used to calculate "J".
5. LO/W/MAJ Line 172: NAME-CHECK is PERFORMED one time too few.
6. PU/E/MIN Line 175: In NAME-CHECK, the check for SPACE is redundant.
7. DE/W/MIN Line 175: the design should allow for the occurrence of a period in a last name.

*A PROBLEM WHICH WOULD CAUSE THE PROGRAM TO MALFUNCTION, A BUG. M=MISSING, W=WRONG, E=EXTRA

Figure 2.4: Inspection Feedback includes: (a) A Code Inspection Report and (b) A List of Documented Issues (Note, this is only a partial list of issues)

2.3.3.2 Formal Inspection Summary

The results found by Fagan's review process showed that for a systems program approximately 66% of errors were found in the inspection processes alone, this is prior to entering the test phase. Another study found Fagan's approach indicated 82% of the errors were found in the inspection process, and so as these errors were found within the inspection stages this meant the testing phase was more of a verification of quality.

Fagan (1976) also presented two major benefits of having an inspection feedback report. First the feedback obtained not only gives the developer an exact location and severity of the issue, but also the type of issue. This then gives an indication of the issue types that mostly occur in ones code (e.g. lack of commenting, design errors, logic errors, performance issues etc.). The inspection report acts as a reminder to the developer in terms of the mistakes they are prone to making. For example at the end of an inspection the user realizes he made a lot of logic errors, being aware of this would increase their caution when it comes to logic coding in the future.

The second benefit of a feedback form is that a ranked list can be generated containing modules with a high percentage of errors. So, not only knowing the majority of error types that occur, but also the location where most occur, indicate the modules that are more error prone then others. This gives the review team an indication of what modules have a high error rate and therefore need extra scrutiny during future development and inspections.

2.3.4 Walkthrough versus Inspection

Walkthroughs differ significantly from inspections, as walkthroughs are informal and incorporate no defined stages. This walkthrough approach simply steps through each line of code and fixes any errors seen. Walkthroughs don't involve any formal procedure, because of this no repeatable results are produced (Fagan, 1976) but having said that they are a more cost efficient approach to code review. They may be cost effective, but according to Ford Motor Company they are not as effective at finding defects when compared to inspections, as 50% more defects were found per 1000 lines of code than what was found during a walkthrough (Wiegers, 2001).

Formal inspections incorporate a more formal approach with its defined inspection stages and specific reviewer roles. Whereas a walkthrough has no preparation stage, and no follow-up stage to verify issues that have been correctly resolved. The code presenter, during a formal inspection is the moderator and is unrelated to the code reader, whereas during a walkthrough the developer takes the dominant presenting role. This may cause problems as the developer may unintentionally (or intentionally) draw attention to specific portions of code and risk overlooking other portions (Wiegers, 2001).

Data from walkthroughs are not collected and stored for future value; issues that arise during a formal inspection are noted within a review report, so can be looked at in the future. Review reports not only state problematic pieces of code but also its severity and class of error. This data can be used for future development projects as it indicates the duration of reviews, review effectiveness, reviewers who actively contribute issues, and the number of errors a developer seems to make. Information can also be learnt for future inspections within the same development project, including the number and type of errors that were found, and the areas of the project that are more prone to errors.

When a project needs reviewing, even though an inspection is more effective at finding errors, walkthroughs are more cost effective. There are two factors that should influence ones decision to adopt a formal review approach and include both the risk and the projects objectives (Wiegers, 2001). Projects can involve risk as they are either difficult to implement and involve complex logic and concepts, or the final project needs to be high quality, as it may be used in a high risk environment where an error can cause severe consequences. In fact quality may be a vital factor in development, as previously a program defect has lead to corporate failure (Arthur, 1993) and even a loss of human life (Leveson, 1993). Projects that are developed by a small organization not yet well-established would be better suited performing walkthroughs. As they can be reviewed by one other person, and so doesn’t take the time away from other developers who also have code to develop. As inspections are more time consuming, performing walkthroughs are more efficient and results in the developer finishing the project earlier and having spare time to take on more development projects.

2.3.5 Code Review Difficulties

As it was mentioned several times throughout this section and by several researchers (Colen, 2001; Fagan, 1976; Johnson, 1994), code review is the most effective means of improving software quality. The results of this process have proven successful and are documented throughout, the results show a remarkable reduction in defects, and an earlier detection of these defects which obviously results in reduced costs in the long-run.

Along with the praise of code review that was mentioned, there are unfortunately several difficulties associated, resulting in code reviews being poorly performed and in some cases even totally neglected. These difficulties must be eliminated, in order to promote code review for software organizations.

The difficulties have been presented in Deimel (1991), Feedman & Weinberg (1990), Gilb & Graham (1993) and Johnson (1994), and can be separated into two groups, difficulties revolving around the manual tasks assigned to the moderator and difficulties in terms of general code review practice. These are summarized below:

Manual Tasks:

Organization

As with any team, organizing can be time consuming, it's no different when selecting a review team. Participants must be chosen based on review inspection experience, their skill level and their experience with related projects. Once the team is selected, they must be notified and the review material must be sent to them, their time schedules must also be known, so an agreement on inspection dates can be reached.

Consolidating individual issues

With some inspection processes (Johnson, 1994), it is common for review participants to first generate their own set of private issues that they want to put forward during the group meeting stage. This puts pressure on the inspection moderator, as they must go through each individual issue from each reviewer and remove any duplicate issues that have already been mentioned from another reviewer. The moderator must then consolidate all issues into a unique list, which they can then present one-by-one during the group meeting.

Generating review report

Once the group meeting is completed and all issues have been recorded and classified, the reviewer's jobs are done. However the moderator must then sort through each issue raised and add them to an inspection review report, which is then given to the developer, such that they can resolve these issues. Filling-out this report by neatly including the issues, along with their severity, line number and description can be extremely time consuming, and adds to the moderator's workload.

With a code review tool integrated directly within an IDE, it gives a computerized approach that can easily eliminate these manual tasks. First, a computerized approach will make it easier to: collect data from previous reviews (in particular, the reviewers present and the number of issues each raised); then choose review participants; then schedule and communicate to each reviewer, all within the same tool. Secondly, if all issues raised by the reviewers can be made directly in the IDE over shared code files, then this allows all raised issues from each reviewer to be viewable by others. A reviewer won't need to raise an issue if they can see the same issue has already been raised.

If issues are raised over private code files, then the ink files created by each reviewer can be collected. The tool can then merge each issue from the corresponding ink files into a unique list of issues based on the annotations line numbers and error-type. Finally, having issues raised directly within the IDE allows annotations to be viewed in the IDE during the group meeting stage. These issues can then be classified in terms of issue type and severity directly within the IDE. Then as each issue is known by the IDE, it allows the moderator to automatically generate a well presented summary of these issues, including the file and line number where these issues reside.

General Code Review Difficulties:

Inaccurate Review Duration

As reviews can be costly, it's important that an accurate overall duration time is achieved. Recording the duration of a group meeting is easy to infer as it can simply be timed from the start to the end of the meeting. However, there are other stages that are more difficult to time, for example both the reviewer's preparation time and the time it takes the developer to resolve all raised issues may be over-exaggerated.

Lack of Preparation

This is one of the major reasons for low quality reviews, and it's caused by some reviewers performing an unsatisfactory preparation stage. This lack of preparation leads to them 'faking it' during the group meeting stage, and being reliant on others to contribute to the review.

Participants Personalities

Personalities can play a dominating role in group interaction environments, either un-intentionally or intentionally, and can restrict a reviewer from feeling comfortable to speak out with an idea or issue. For instance, the moderator may present an unpleasant persona, where an inexperienced participant feels intimidated to raise an issue or idea. One reviewer may feel uncomfortable when criticizing someone else's work, especially if there is an element of doubt. Finally, participant's personalities may clash in a decision that sparks a debate, which may cause conflict and carry through to other raised issues.

Recording Difficulties

Accurately denoting each issue while reviewing code can be more difficult than expected, as good descriptive notes must be written efficiently. There are three possible approaches. First, annotating over a printed out version, this worked well when code was written sequentially. However now code is object oriented and separated into object classes and procedures, and read in a non-linear fashion.

Second, reading code from the IDE then documenting the issues in a text editor or on paper. This is unsuitable as noting an issue over a separate document can be distracting and risks a break in attention as one switches between these applications (Marshall, 1997; O'Hara & Sellen, 1997). It is also recommended that each comment must be attached with a direct reference (O'Hara & Sellen, 1997) and be in close proximity to this reference (Atkinson et al, 2000). Third, writing the issue directly in the IDE as a comment line results in issues being 'hidden' amongst code, and so each comment must be scrutinized to determine if the comment relates to an issue or a simple code comment.

Johnson (1994) investigated ways to eliminate these general code review difficulties. Their approach was to amend Fagan's code review approach (Fagan, 1976) by splitting the preparation into two stages a private review and public review. The private review allows each reviewer to generate their own issues, and then all issues generated can be viewed by all during the public review stage. The private review ensured each reviewer did an adequate amount of preparation and couldn't 'bluff' their way through the review, as the moderator could quickly view all the issues made by each reviewer. This private and public review approach also eliminated most conflicts between reviewers and the moderator. On the basis of issues now being raised during the private review, reviewers have plenty of time to word their ideas and issues in a non-intimidating environment. As issues are already raised before the group meeting stage, most of the time is used to simply to iterate through known issues rather than spending time looking for new issues. Johnson (1994) also used an intelligent timer to get an accurate estimation on the duration each reviewer spent reading code. This timer resides in a code reading tool, and only counts each minute if there was a sign of user interaction within this tool e.g. scrolling, mouse movements or keyboard actions.

The last difficulty mentioned involved problems when recording issues in a convenient manner making sure to include a direct reference in close proximity to the code. Explaining exactly where to find an issue within a separate text editor may give a good direct reference point to the issue, if the issue is well described; however the raised issue would not be within close proximity of this reference. Our research of supporting digital ink annotation directly within an IDE, will give a more effective and efficient approach for reviewers to record issues, nearby to the issues reference point.

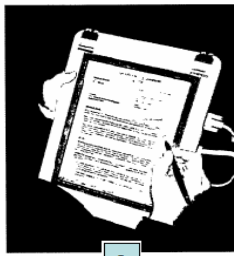
2.4 RELATED ANNOTATION TOOLS

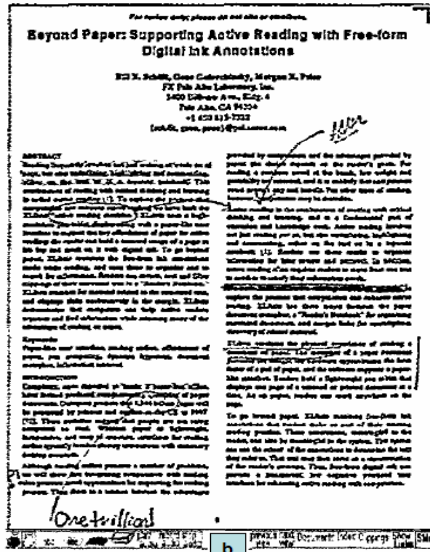
The tools mentioned in this section allow the user to supply feedback directly over several different types of documents e.g. text documents, web documents, student assignments or coded documents. Schilit et al. (1998) developed 'Xlibris' that allowed readers to review textual documents by supporting digital ink annotation, an extension to 'Xlibris' was made by Golovchinsky & Denoue (2002) and allowed the document to reflow its annotations as it's layout changes, and Ramachandran & Kashi (2003) demonstrated how to incorporate digital ink into an existing web browser for reviewing web documents with digital ink annotation. As an ink annotation tool within an IDE can provide marking capabilities for coding assignments in the future, both the 'MarkTool' (Heinrich & Lawn, 2004) and 'Penmarked' (Plimmer & Mason, 2006) offer some interesting annotation concepts, whereby each annotation tool takes a separate approach for providing feedback and supplying grades/marks directly over student assignments. All these tools mentioned above can be used as part of active reading where key ideas (personal ideas) and review feedback (where someone else gives the author feedback) can be supplied.

B. N. Schilit, G. Golovchinsky and M. N. Price
(Beyond Paper: Supporting Active Reading with Free Form Digital Ink Annotations)

Schilit et al. (1998) describes one of the earliest tools to incorporate digital ink annotations on a graphics tablet. The tool is called 'XLibris', and was implemented in C++ to support active reading over a natural paper-like user interface. 'XLibris' allows ink annotations over 'static' PDF documents, meaning its content and layout is not editable (fixed). Due to the document being static the ink annotations won't need to move (reflow) to different locations. The key goal with this tool is to provide a digital environment that supports the tangibility benefits and easy page orientation that are both seen in traditional pen and paper annotation.

Tangibility refers to the physical properties of paper documents, such as being easily held, tilted and rotated, which is not readily available when using large desktop computers. Simply designing an active reading tool that uses a light-weight tablet display with the capabilities of digital ink, gives a more tangible reading experience. The tablet also includes physical widgets built onto the top of tablet unit shown in figure 2.5(a). Viewing a new page now mimics a book where a thumb-press of a button turns the page back or forwards, rather than scrolling through the document or tapping a button with the stylus.





Beyond Paper: Supporting Active Reading with Free-form Digital Ink Annotations

FX Palo Alto Laboratory, Inc.

Figure 2.5: XLibris User interface includes: (a) Tablet Display, (b) An Annotated Document in Xlibris, (c) The Document-View

Page orientation of paper documents offer users good visual feedback of the length for each page, as an entire page can be viewed at one time. This allows the user to quickly flick through pages, and can use a particular figure/table or a recognized page layout as an indication that the required paragraph is close-by. The thickness of a document can also be approximated and used to find the approximate location of a certain passage (location guides). ‘Xlibris’ provides location guide indicators as a way of giving readers feedback about the documents thickness, as well as their current location within the document. An annotated document in ‘Xlibris’ is shown in figure 2.5(b) along with its length indicator in the top right corner and its thin unobtrusive tool menu along the bottom. The tool also contains added functionality, for instance it provides a ‘document-view’ of the currently annotated PDF, shown in figure 2.5(c). The ‘document-view’ displays small thumbnail representations of each page in the document and serves readers by giving them a subtle representation of pages to come, as well as providing quick navigation between pages. These small thumbnails are illegible, but they do indicate the basic representation of a pages layout, and its figures, and its digital ink annotations.

As it is common for active readers to work with several related documents simultaneously, they tend to scatter them across their desk so they can quickly select a required document. ‘XLibris’ caters for this situation by allowing the reader to load in several documents into the tool then use the ‘workspace-view’. This display shows the first page of each document as thumbnails, similar to the ‘document-view’ and helps users navigate between different documents.

G. Golovchinsky and L. Denoue

(Moving Markup: Repositioning Freeform Annotations)

Golovchinsky & Denoue (2002) describes an extension to the previously mentioned tool 'XLibris' (Schilit et al., 1998), by allowing greater flexibility for document annotations. The earlier version revolved around ink annotations over an underlying static PDF document, where annotations would be permanently fixed to page coordinates. However this extension provides a solution to the fixed layout limitation by providing an algorithm to reposition and resize existing ink annotations when the layout of underlying documents changes. This algorithm maintains the correct positions of the digital ink throughout a documents font changes (style and size), changes in aspect ratios (e.g. zooming in/out, changes in window size) and changes in device type (with different display properties).

The challenging part of repositioning the ink strokes occurs when annotations resembling underlined and circled passages need reshaping as their underlying text splits over to a new line (line break) or over a new page (page break). When the documents layout is changed the tool extension reflows the ink annotations to a new location based on the new layout of the document, the type of annotation and its original location and size. The algorithm described in this paper contains reflow approaches for three different types of ink annotation:

Annotations attached to a line of text

For annotations that are attached to a series of words (e.g. underlines, highlighting and circled annotations on the same line) then the algorithm splits the original annotation into individual segments. For example if the annotation covers five words all on the same line, then the original annotation will be split into five overlapping bounding boxes. An individual box covers a single word and these boxes are linked together making it easier to handle reflow when the words of the document move to a new line.

Margin bars and circled passages

Unlike annotations attached to lines, these annotations don't refer to individual words, rather they refer to a range of words on multiple lines. During a layout change, these annotations may need to be vertically stretched/compressed as the text may flow over more/less lines. To support this situation the extension will create a bounding box for this annotation along with a two anchors corresponding to the first and last word within the passages bounding box. As the first and last words move, the location of these words can be used to determine the extent to which the ink is stretched out.

Handwritten notes and symbols

These annotations are dealt with in a similar manner to margin bars and circled passages with the only difference being the bounding box is only a vertical location identifier. This annotation is attached to the word alongside the top of the annotations bounding box and won't be stretched when the document is repagination, instead theses annotations can only relocated. However future investigations of this paper will determine whether these annotations will be scaled down when displayed on smaller devices.

The evaluation of this extension described a major fall-out of this algorithm and occurs when repositioning handwritten text. This flaw is caused as some users print words, and so an individual stroke is used for each letter, rather than using a lower number of strokes for their handwriting. This exhaustive number of ink strokes resulted in some difficulty when classifying the stroke into these three types of annotations. To minimize this problem, ink annotations were grouped into logical units based on both time and the annotations location. This approach is consistent with Bargeron & Moscovich (2003) and was considered an important factor when improving this algorithm.

When grouping strokes by time, the data obtained from the initial evaluation of Golovchinsky & Denoue (2002) shows that the time between strokes peak at around 250ms, then as this time increases the percentages of strokes belonging to other words also increased. For example annotations within 500ms of each other only 11% of these strokes belong to different words, so as time progresses this percentage increases. The data also shows that future annotations made near existing words also correspond to the same word. This indicates that the annotations spatial characteristics should also be considered when grouping annotations and that time alone is not a good grouping indicator (Golovchinsky & Denoue, 2002).

S. Ramachandran and R. Kashi
(An Architecture for Ink Annotations on Web Documents)

Ramachandran & Kashi (2003) describes a web tool prototype that captures, then renders and attaches the digital ink strokes to the underlying elements of a web document. This system shown in figure 2.6, handles digital ink in two ways: as free-from annotations that are part of the document; or as a 'pen gesture', where the system decides what action to take. In order to allow the digital ink to interact to changes within the underlying web browser the tool uses both the W3C's Document Object Model (W3C, 2005) and Dynamic HTML (DHTML) to map ink strokes to web text and images.

This system can therefore be used by web developers as a way to critically evaluate and supply feedback to a recently implemented web page, or even as an active reading tool for the public to capture key ideas from the web content. This tool stands-out from previous annotation tools (iMarkup, 2006; Schilit et al., 1998) as it successfully reflows its digital ink to support any possible changes (from server-side) in both the documents layout and content. This is done by attaching the ink strokes to the web element tags, so as the web element moves to a new location, this location is shown within the elements tag and so annotation’s can move to the same location.

Figure 2.6: Digital Ink Annotation over Internet Explorer

This web document annotation system supports loading and saving the ink annotations (along with attributes and properties) on the client’s hard-drive. The system uses an XML schema (W3C, 2006) which allows for extra attributes and properties to be stored along with the rendered ink and its location. Storing extra attributes along with the ink strokes is important as it can allow ink annotations to contain additional information to provide further functionality. For instance, as soon as a URL is loaded a search is made for any the associated ink on the client-side that is based on the same URL, and if found it would be overlaid on the underlying web document. As both ink and its attached web element tags are stored together, if the loaded web page has been changed slightly on the server-side, then the digital ink can be adjusted to accommodate.

The system also takes advantage of a 'hysteresis smoothing filter' (Duda & Hart, 1973) which reduces jilter of the mouse or stylus's ink coordinates when annotating. The system uses keyboard modifiers (ctrl, alt, and shift keys) to switch between annotation mode (free-form annotations) and recognition mode (pen gestures). When in the recognition mode the system recognizes pen gestures with between 98 - 100% accuracy. For example, the 'left arrow' gesture is used to reconstruct a new partial HTML file containing only selected HTML fragments, and the 'up arrow' and 'down arrow' indicates scroll up and move back in history respectively.

When in the annotation mode, a range of text within the HTML page can contain free-form ink strokes, they can also be selected then bolded or highlighted, even columns within tables can also be selected or annotated.

E. Heinrich and A. Lawn
(Onscreen Marking Support for Formative Assessment)

Heinrich & Lawn (2004) investigated how one can utilize computer technology to support the marking of student assignments, and provide quality feedback over their submissions. The aim was to replicate the tasks seen with traditional pen-and-paper assignment marking and also incorporate the strengths of computer technology into an onscreen marking and annotation tool called 'MarkTool'.

'MarkTool' is shown in figure 2.7, and at its time of development was one of the first of its kind, and seen to provide a positive student learning experience. The current versions of Adobe Acrobat (Adobe, 2003) and Jaws PDF Editor (Global Graphics, 2003) represent comments in note-form and are do not offer clear feedback annotations, both applications fail to support any type of marking capabilities (Heinrich & Lawn, 2004) or priority management. So it was decided to implement this functionality as a separate tool, as feedback comments and marking were seen as the most important aspect of this tool.
The application obtains assignments in PDF format via communicating with the class assignment database. It then allows the marker to highlight an area needing feedback (using a line, box, circle or a free-hand boundary), and then they can attach type-text comments that are linked with the highlighted area. So the tool allows the user to provide a further description alongside their highlighted annotations.

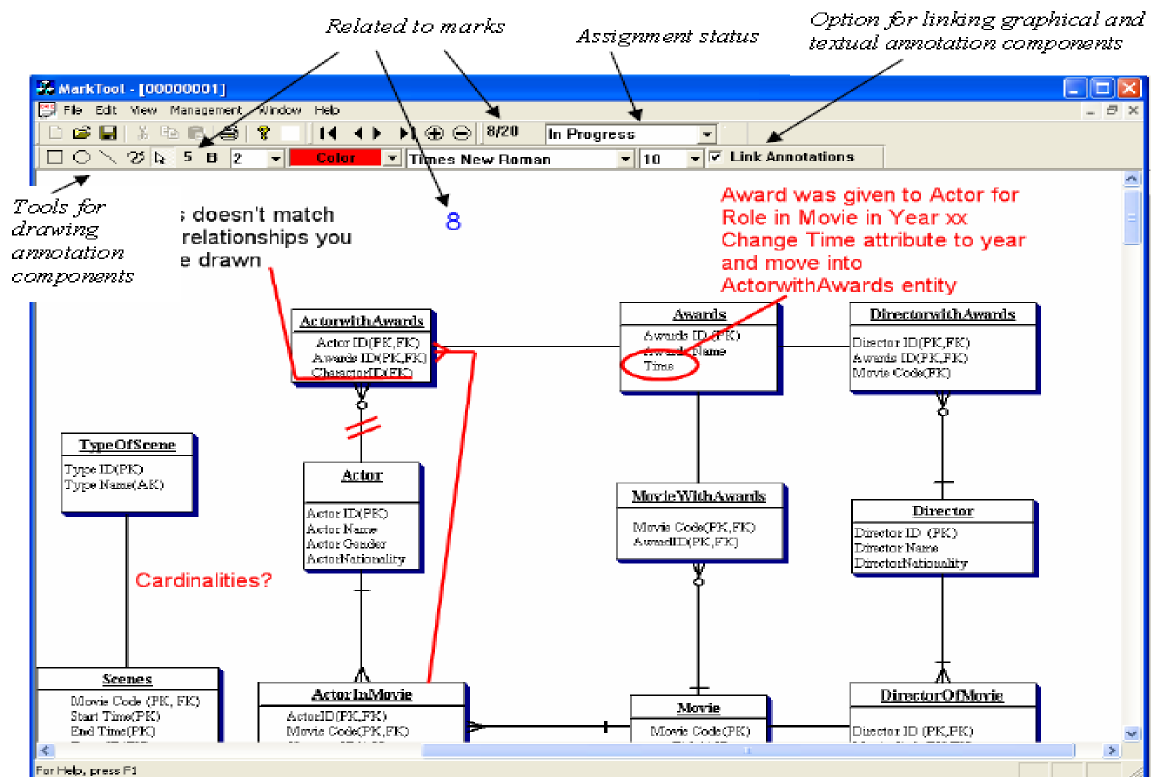


Figure 2.7: MarkTool User Interface

It was decided to represent annotations using a graphical component to indicate the exact area an annotation would refer to, and a text component describing the annotation followed by a link between the both. It was also agreed that different colours and fonts may be required for both graphical and textual components, as they provide different categories of feedback, suitable editing (copy, paste, delete and resize) was also thought to be useful for both components.

The approach to record marks on these digital documents was met with some difficulty as the style of mark, can be represented in several ways and placed anywhere over the document. For example, one out of four could be represented as: ticks, 1/4, 1, or -3. So it was decided to take a minimalist approach, where the overall possible mark and the mark given for the assignment were displayed in the toolbar. The tool comes with a 'marking progress-status', as multiple assignments can be opened, the status indicates to the marker the assignments that have already been marked and what assignments still need marking.

The evaluation of 'MarkTool' tested how well it performed against traditional marking methods, and by its convenience. The results were positive, with people finding typed comments were "faster and less strenuous and facilitates copy & paste" (Heinrich & Lawn, 2004). Another pleasing aspect of this tool was the annotation comments being easily modified during the marking process.

B. Plimmer and P. Mason

(A Pen-based Paperless Environment for Annotating and Marking Student Assignments)

Plimmer & Mason (2006) developed a tool called Penmarked, a paperless marking and grading environment for student assignments, written in ‘C#’. This tool allows the marker to obtain a collection of student assignments, and then supply feedback directly over the student’s original submissions with digital ink strokes. Once all assignments are marked they can be emailed back to the students directly through the Penmarked tool. Figure 2.8 shows the standard interface layout of the Penmarked tool, with its three main components (a) a list of student submissions, (b) the annotation pane and (c) the marking schedule.

Once the tool loads a group of assignment submissions and the assignment marking schedule, the marker can then view individual assignments by selecting it from the list of student submissions (figure 2.8a). As each submission can potentially include several files, they are all viewable within a multi-file interface seen in the annotation pane (figure 2.8b), where tabs are used to view a specific file (similar to file tabs within an IDE). The system is designed to show source code assignments, however it could be used to annotate any type of assignment independent of the topic. Once the assignment is displayed the user can then use the power of ink annotation to make digital markings directly over the opened assignment. The marker can also open and use the IDE as a separate process to compile and test the assignment.

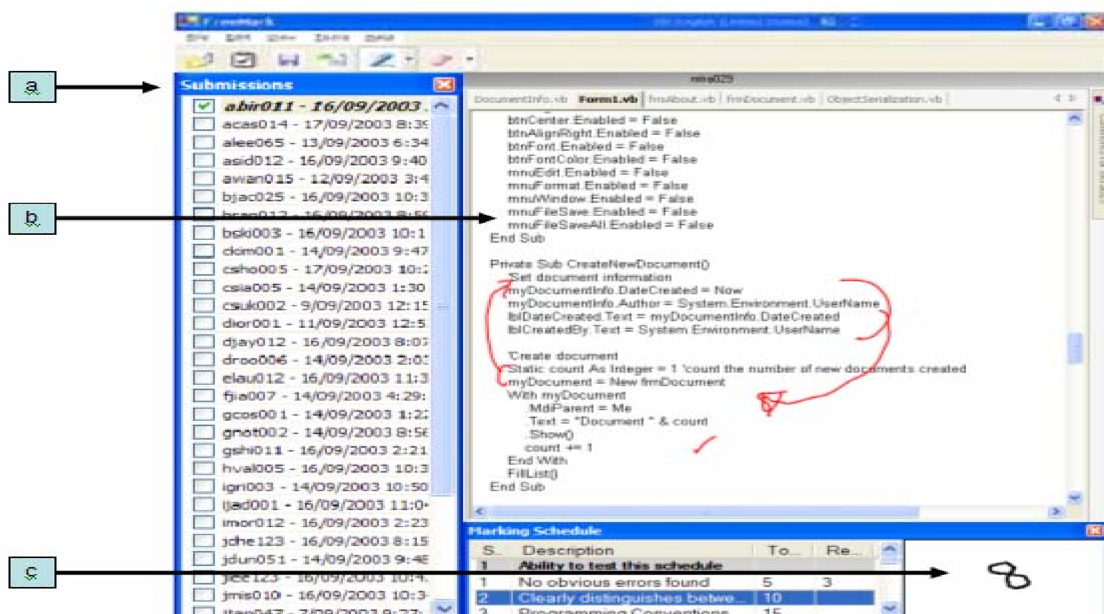


Figure 2.8: Penmarked User Interface includes: (a) Student’s Submission List, (b) The Annotation Pane, (c) Marking Schedule Design

A marking schedule (figure 2.8c) is used by the system to grade student assignments. This schedule contains a list of marking criteria along with the minimum and maximum values that can be obtained. Once the marker selects a specific criterion to mark they can write a number (between the minimum and maximum value) into the marking recognition pane and have this score recognized. After a time-limit has elapsed or if another mark criteria is selected, then the marking pane may flash green signifying the inputted mark was successfully recognized. If the score is recognized, then it's entered as the value for that selected marking criteria. However, if the mark validation failed, then this is indicated by the marking pane flashing red, which acts as a cue to the marker that no mark was entered into the schedule.

Once the marker has finished grading and scoring an assignment its progress status changes to 'complete'. This is shown in the student submissions list, and used to indicate the assignments that have already been marked. All the ink files are then saved with the submission files and emailed back to the student in PDF format. All assignment scores are saved in one overall XML (W3C, 2006) formatted file which can later be sent to a database or stored in the course spreadsheet.

2.5 Related Integration Tools

This section gives a basic overview of the three integrated tools and its architecture, and then describes how the added functionality gives convenient benefits to their users. The first tool is integrated into the 'Eclipse IDE' (The Eclipse Foundation, 2006), and provides the user with a more efficient approach to search for code dependences within the code project. The second tool was integrated into 'Visual Basic 6' (Microsoft Visual Basic, 2006), and provides novice programmers with an easier approach to design windows forms in a shareable manner. The third tool was integrated in 'Visual Studio 2005' (Microsoft Visual Studio, 2005) and integrates a tool into the code framework, such that the user can code and view new 'blogs' within the same application.

M. P. Robillard and G. C. Murphy
(FEAT: A Tool for Locating, Describing, and Analyzing Concerns in Source Code)

Robillard & Murphy (2002) developed an Eclipse plug-in to provide a different perspective for code fragments (class, function or variable) that are of interest to the developer or code reviewer. For instance, consider a program that includes 100 code files, and someone with the task of resolving a logic error where a variable contains an incorrect value. The reviewer would have to back-track throughout the code

to find all variables that influenced the erroneous value which is a tiresome and frustrating task. The 'FEAT' plug-in solves this dilemma by integrating a code analyzing tool seamlessly in the Eclipse IDE, as shown in figure 2.9. The IDE performs as usual, however it gives extra functionality to represent all dependencies of a selected code element (class, function or variable).

The FEAT architecture consists of three major components: a 'concern model' that provides the operations on code concerns; a 'byte-code analyzer' provides relations between different elements in the source code; and the two GUI components (in particular a 'projection view' and a 'participant's view') that is incorporated into the Eclipse Platform as two separate windows.

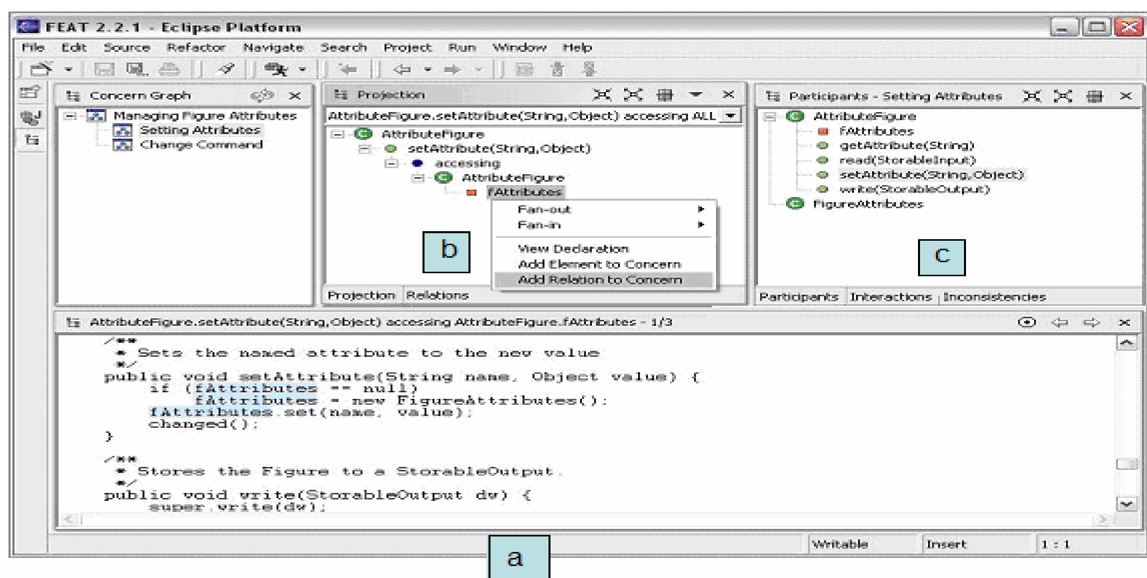


Figure 2.9: FEAT User Interface in the Eclipse Platform includes: (a) Code Window, (b) Projection View, (c) Participants View

The tool allows any element from within the code window (shown in figure 2.9a) to be selected and shown in the 'Projection view' of the FEAT window. The 'projection view', shown in figure 2.9b is essentially a list of elements that the user wants to investigate (a concern). Any element in the 'projection view' can be moved to the 'participants 'view'. The 'participants view' shown in figure 2.9c, shows only the elements in the program that are dependant on the selected element (e.g. the variables that participate in the selected elements value) and is represented similar to a declaration tree in Eclipse. The users can also view the portion of code that corresponds to any element in either the projection or participation view, this code portion is shown in the code window, and is clearly highlighted.

B. Plimmer and M. Apperley

(Freeform: A Tool for Sketching Form Designs)

Plimmer & Apperley (2003) developed an ink annotation sketch tool called 'Freeform.' 'Freeform' was designed for novice programmers to create the design layout for their application forms, performed using ink annotation rather than coding or using the IDE design window. Once the design is drawn with digital ink over the sketch interface, the user can have their informal sketch layouts translated into formal designs. This sketch tool was developed as a Visual Basic 6 (VB6) add-in, and allows the user to easily move between their informal layouts and the frameworks version of the formal design.
The underlying functionality of the Visual Basic IDE works as usual, but with an extra layer of functionality. So, this sketch tool resides within the same environment as the IDE, and has the ability to interact directly with the VB6 framework.

The tool was primarily developed for use with an interactive digital whiteboard shown in figure 2.10a, as it provides a large shared group workspace (Plimmer & Apperley, 2003). However it also works well on a portable Tablet PC where form designs can be informally jotted down 'on-the-run' or during a coffee meeting with a client. Once the tool add-in is loaded it presents the user with a sketching interface shown in figure 2.10b, where form components can then be drawn onto this interface using specific shapes. For instance a square box sketch would translate to a text-box, whereas a small triangle within the right-hand side of a square box would translate into a drop-down list.

The tool incorporates two inking modes, a 'drawing mode' where strokes are recognized as shapes, and 'writing mode' used to write words. An 'editing mode' is also provided for copying, moving, or resizing a selected shape or group of selected shapes. Ink strokes can also be changed from a drawing annotation to a writing annotation and vice versa, an infinite undo stack is also available to the user so their actions can be individually reversed. Once the user is happy with their informal design, they click the 'map' button on the sketch space, this revels how the form will look if committed. Rubine's recognition algorithm (Rubine, 1991) is then performed, which attempts to recognize the informal shapes, and translates them into VB form controls (based on a shape to control mapping). The controls are fixed to a grid with all controls having a unit height and width value. The user can also edit any control where a recognition error occurred and once they are satisfied with the translation then 'Freeform' will generate the VB form as seen in figure 2.10c.

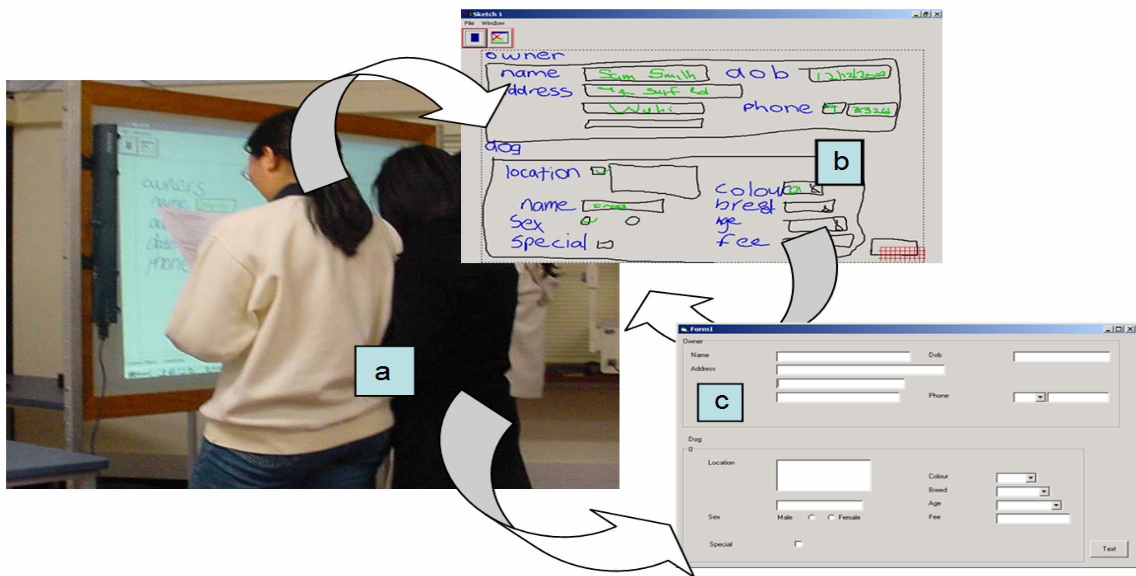


Figure 2.10: Sketching User Interface Designs in Freeform includes: (a) Visual Display, (b) Informal Design Sketch Space, (c) Formal Design

'Freeform' is capable of translating an informal design to a formal representation, and then generate the VB form. Not only that, it can also dynamically list a set of attributes associated to a specified control, and allows navigational links to be added between one form and another destination form. An evaluation of this tool was performed and met with a positive attitude, with the participants finding the tool a joy to use. As this tool was implemented as a VB add-in, it shows that VB6 is more than capable to combine additional functionality into the existing framework. Plimmer & Apperley (2003) mentioned that other IDE's with similar add-in capabilities could also incorporate the functionalities of 'Freeform'.

J. Conwell

(Blog Reader Add-In for Visual Studio .NET 2005)

Conwell (2004) gives an in-depth description of a tool developed and integrated into Visual Studio 2005 (VS2005) called 'Blog Reader' which is shown in figure 2.11. This describes the experiences and the difficulties faced by a professional developer when integrating a tool into VS2005. The reference also provides full implementation details and a downloadable copy of this VS2005 add-in. We present this tool in this related work section as it is a great code example for anyone who is interested in creating a VS2005 add-in project.

This tool was not used to give the developer extra programming functionality in the IDE, in fact this tool is totally un-related to programming. The ‘Blog Reader’ was simply integrated into the IDE as the developer spends most of the day programming in Visual Studio as well as regularly checking posts on several registered blog sites. This developer found that switching between different blog sites to check for new posts while also coding in Visual Studio involved too much manual effort. So now blogs can be viewed and projects can be coded all within the same environment.

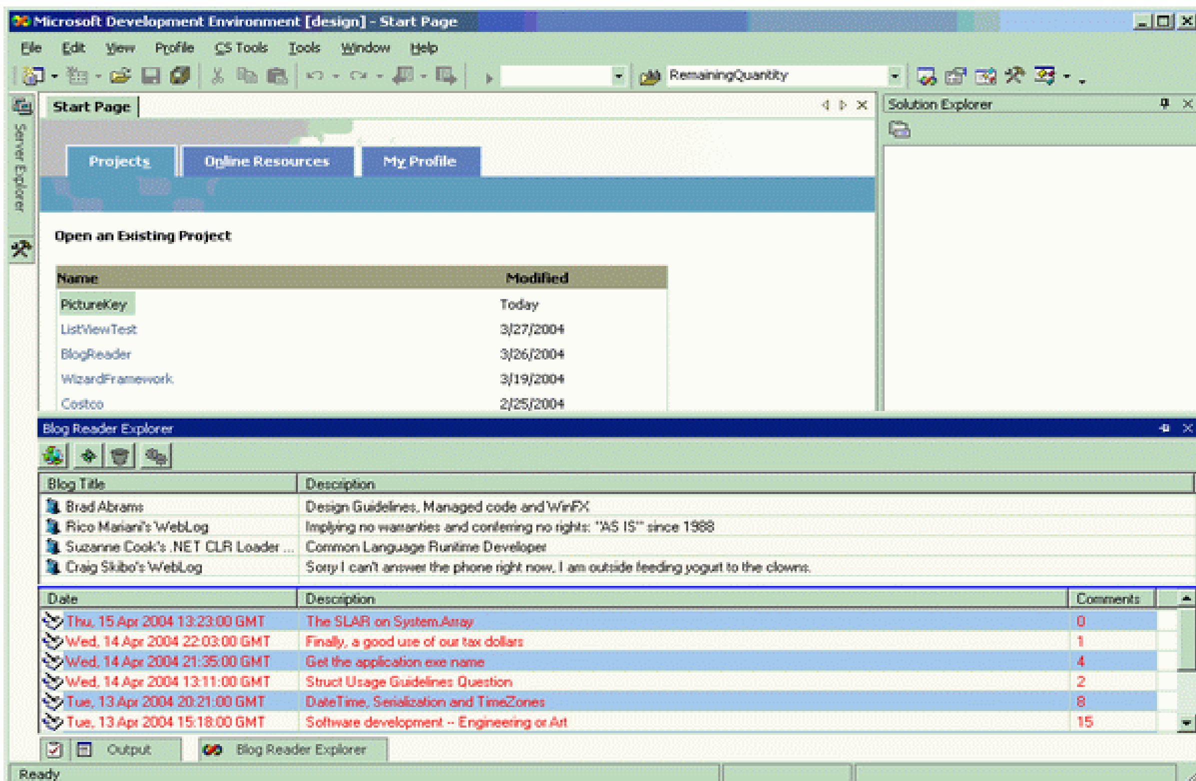


Figure 2.11: Blog Reader Explorer Interface in Visual Studio 2005

The add-in works by automatically downloading the latest blog entries from all the blog sites he had subscribed to, and is initiated as soon as the IDE is loaded. Each blog would also include the title, publish time and will be highlighted if the entry is new. The comments for a specific blog entry can be read by selecting the ‘View Entry Comments’ option, and selecting the ‘View Whole Blog’ option will open the blog’s homepage.

The ‘Blog Reader’ was integrated as its own window object that either remains ‘docked’ to the IDE or as a floating window, similar to docking and un-docking other windows within the framework. A tool window can be created in Visual Studio, however can only host an ‘activeX’ control, which unfortunately can not be developed in C# or VB.NET. The ‘Shim’ control found at Yahoo! Group News (2006) is essentially an activeX control developed in C++, and once downloaded can be used to create this window object. Once this tool window is created with this ‘Shim’ control a user control (similar to a windows form) can be easily designed to contain window components and then added to this tool window.

2.6 SUMMARY

In this chapter we covered the background material from existing ink annotation and code review research. After giving a brief overview of these key concepts a review of some related tools was presented, where annotator feedback was supported directly over different types of digital documents. To present the integration capabilities available with existing frameworks, we also looked at tools that were seamlessly integrated into three common IDES’s.

The concept of providing ink annotation within an IDE to support industry code review also holds true for other types of code review. First, in the classroom as the teacher explains portions of code and uses annotation to emphasize particular points. Second, the tool can be used by developers to comment over their own code, much as they would with keyboard comments. Third, when marking students programming assignments, the marker can indicate where the student went wrong in their code and give an overall grade.

Integration of a code review tool into an IDE could support all the benefits of the previously mentioned code review concepts (Fagan, 1976; Johnson, 1994), and the ideas from the ‘Penmarked’ marking tool (Plimmer & Mason, 2006). The tool would also retain all benefits the underlying code editor has to offer, such as project debugging and running, object-definition searches, font properties and existing wizards.

Chapter 3

Scenario of Use

3.1 INTRODUCTION

This chapter presents design specifications for an optimal solution of an IDE ink annotation tool by incorporating ideas from different code review scenarios. Scenarios were employed as they are a proven technique for generating an exhaustive list of user requirements (Rosson & Carroll, 2002). To decide what this tool must incorporate, we describe scenarios of five types of code review as well as how they would use an IDE ink annotation tool to substitute their common code review practice. These scenarios are teaching a class, marking assignments, industry-based formal review procedures, peer reviews and as a commenting tool for developers. With an understanding of the types of tasks performed by each of these code reviewers, we can select a list of research ideas from the previous section that would give additional code review support for an IDE annotation tool. Concluding this chapter is a description on the final requirements of an IDE code review tool in its optimal state.

3.2 TEACHING IN THE CLASSROOM

Scenario

In an academic computer science environment teaching a new programming language to students is a prominent task. It involves a lecturer either preparing screen shots of code on handouts, or providing written code within an IDE where it can be viewed and executed. The prepared code is then shown to the students in class, where the lecturer explains the source code and the students can write both the lecturers notes and their own additional notes on their handout.

During the class, the lecturer has several devices available to show the code to students. One might use the traditional blackboard and chalk, or the modern white board and marker, an overhead projector is another commenting device, where the code is projected onto a white sheet.

Either way the notes made by the lecturer to describe code are not preserved for future reference; therefore students may spend most of their time writing these notes onto their handouts for future reading. Rather than write notes continuously during class that reproduce the lecturers material, students could spend more time listening to the lecturer and only writing down their key ideas.

After the class, the coded examples used by the lecturer are generally stored on the course web site, however all corresponding notes for the examples written on the board or overhead are lost. The only way the lecturer can write notes directly on the code during the lecture are via comment structures (e.g. '//' in C#) which can be time consuming and means the lecturer has to sit at a computer to type this, rather than standing in front of class. Furthermore these comment structures don't standout from the code so can be difficult to locate when quickly scrolling through large amounts of code.

Solution

This ink annotation tool can be seen as a benefit for lecturers both during and after the lecture. Before the class the lecturer would instead, simply provide the handouts as done previously, however all coded examples would be implemented within an IDE. As the tool is designed for a tablet PC, an overhead projector can be used to display the code example on the whiteboard for all to see. During the class, the lecturer can explain the coded example by using digital annotations directly over the code, rather than writing comments on a whiteboard or an overhead device and obstructing the board. Because the annotation tool is seamlessly integrated into the IDE, the digitally annotated code can still be directly compiled and executed. After class, not only the code but also its corresponding annotations can be stored on the course web site, therefore the lecturer's whiteboard notes that would be previously erased are no longer lost and can be viewed at a later date by the student and teacher.

3.3 MARKING

Scenario

In an academic computer science environment, marking assignments is frequently performed. With an increase of computers used in universities, the majority of these assignments are generally submitted online. The procedure of marking computer science assignments begins by students uploading their work and personal details to the lecturer or to an assignment drop-box (Department of Computer Science, 2006).

The marker gets access to the assignment submissions, and one-by-one views each assignment project ensuring the program compiles correctly, and then they check the assignments content against a certain set of specifications. The marker would make a new text file copy that represents the student's mark-sheet, this contains a marking schedule checklist of requirements with a place to insert comments and a mark for each requirement, along with an overall mark. Code portions of the assignment are viewed ensuring these segments of code meet the correct specifications of the marking schedule. Comments regarding the success or failure of meeting these specification requirements are noted in this text file along with their associated mark. Once all sections of the marking schedule are filled out, the marker sums up the overall score for the assignment and writes this overall mark at the bottom of the marking schedule.

The overall mark is then stored in the course mark database or spreadsheet file and the marking-sheet is emailed back to the appropriate student. The problem with this approach is that the project is compiled and executed using an IDE and so a break in the task occurs as the marker switches between the IDE and the text editor. There is also no direct point of reference between comments in the marking schedule and the related portion of code. The marks must be manually written into in a spreadsheet database, and then emailed back to the student, again this is all done via separate applications.

Solution

An ink annotation tool within a coding environment allows the marker to handle each aspect of marking within the one application. Similar to the ideas in Plimmer & Mason (2006), the tool would load multiple assignment projects from within a directory, rather than the marker loading each project one-by-one. However as the tool is integrated into the IDE, the tool can also compile and execute individual projects. The tool can load a specific marking schedule that relates to a specific type of assignment which the marker can fill-out. With digital ink support directly over code, the marker can also give the students positive feedback, and comments as to why marks were lost with a direct point of reference between comment and code. More importantly, markers can make non-textual comments, near portions of text such as vertical-bars and circled passages, ticks and crosses, question-marks, smiley-faces. This informality can not be done through the previous assignment marking approach, unless the digital copy is printed out. Marks for each requirement can be given in the marking schedule with digital ink, and an ink recognition engine can interpret these as integers and add them to the overall mark.

Once the marking schedule is filled out, the tool can automatically email the students code, along with their digital ink comments, and a read-only version of the marking schedule. The student can then view the schedule and place the ink files within their assignment directory which can then be viewed when they open their project.

3.4 Formal Technical Review Procedures

Scenario

Within the development industry, organizations may use some sort of formal technical review (FTR) process to ensure high quality software. The first stage of an inspection allows the review participants to prepare as they are given the artefact before the group meeting. In some cases, the preparation is split into a private review followed by a public review (Johnson, 1994). During the private review each participant generates their own issues on paper or within a text-editor and is for their eyes only. At the end of this stage, each reviewer gives their list of issues to the moderator, who uses them to manually consolidate a unique list of issues for all to see. This is an effective approach as it gives the moderator high quality feedback of each participant's level of preparation and understanding. The reviewers can then comment on issues generated by other reviewers in the public review stage. After the preparation reviews, the group review meeting becomes more efficient as most issues have already been discovered, they just need classifying.

This formal procedure in most instances ends with a group review meeting, where the reviewed code is either printed out and a copy given to each participant, or the reviewed code would be displayed using an overhead projector such that all can see. Fagan (1976) suggests the group of participants is best limited to four or five technical staff (including the developer) together with the moderator whose job it is to control the meeting progression (this is usually the project manager). During the meeting, the participants iterate through the code, indicating portions or lines that represent an issue and if issues were noted during preparation these would also be covered. When an issue is found the line number, type of issue and its severity is decided upon, this information is then recoded by the moderator.

At the end of the review meeting all issues are manually recorded in a 'review report' which is then referred to the developer to view and resolve the issues. This report contains the line numbers of the issues, such that the developer can easily navigate to the appropriate issue, as well as the issues severity, such that the developer can prioritize the order of issues, and finally the type of error made, to give the developer a better understanding of the issue and an inform them of the majority of errors-types made.

Solution

If an organization includes some preparation stage within their technical review procedures, then perhaps incorporating a timer as explained by Johnson (1994) to show the time a reviewer actually spent reviewing code. As the review is conducted within a computerized tool, a check would be made to see if the reviewer interacted with the code material before the minute is counted, this interaction can come from scrolling the document, or making an annotation. The timer gives a more accurate indication of the

preparation time, rather than the reviewer including the time taken to check emails, talk on the phone and eat lunch. This would therefore eliminate any possibility of the reviewer over exaggerating their preparation time.

For technical review procedures that involve individual private reviews the reviewer can use the ink annotation tool to record issues. After the private review, all ink files can be sent to the moderator, they can have the ink annotation tool automatically merge all ink issues into a unique list. This is done by eliminating similar issues (from multiple copies of the same code file) that appear on the same line, and are the same error-type. The ink strokes can then be incorporated into a shared code project where all reviewers can view this new list of issues. The author of each issue can be stored with each annotation, such that any questions with certain issues during the public review can be clarified by discussing it with the annotation author.

Within the group meeting phase, this tool can work well with a tablet PC and an overhead projector. Similar to projecting code onto a screen in the teaching scenario, the moderator can easily write issues using digital annotations on the tablet as they are raised by the review team, again without obstructing the projected code. The tool should also be capable of attaching individual severity ratings for each issue, and can use different ink colour to represent different issue types. At the conclusion of the group meeting the tool should automatically generate a review report containing each noted issue, along with the file and line numbers, severities and issue types in a formatted text document.

Whether the review is in the initial preparation stage or within the group meeting phase, having issues made directly over the portion of code with digital ink allows for clearer and a more noticeable representation. Allowing the ink annotations over a developer's code also makes it easier and quicker to generate and display issues and with computation support makes it more efficient to generate review reports and merge similar annotations. This tool would make formal technical review procedures more widely and efficiently used.

3.5 SELF/PEER REVIEWS

Scenario

Self/Peer reviews are essentially walkthroughs, where the developer or their peers take a detailed line-by-line tour of the coded project (Colen, 2001). The difference between self reviews and peer reviews is during a self review the developer conducts a walkthrough by fully comprehending the code, much like

describing it to someone else. It’s the idea that when explaining your code to someone else, issues become more apparent and noticeable. Eli Weber an experienced software developer once said “If I could talk to the wall as if it were a person, I wouldn’t need someone else in order to do code review” (Colen, 2001).

A peer review involves a mentor, a friend or a work colleague to walkthrough your code and note possible issues, similar to handing code to the company software testers. Having someone else review code may be more effective because they generally see issues that the writer was unable to see, much like having someone else proof read your essay. Like formal inspections, the reviewer can either mark issues directly on the printed-out version of the code, or the reviewer can read the code in an IDE and use separate paper to represent where the issues are located. After the review, the reviewer gives a copy of all raised issues to the developer to resolve.

The problem with having one of your peer’s review your code is that they like to fix errors, even though they are generally unaware of the underlying context of the software and so results in being an inefficient use of the reviewers time (Fagan, 1976). Not having a full understanding of the code can cause problems when attempting to change someone else’s piece of code, as one change may have an unanticipated impact on several other portions of code.

Solution

During a self review, the developer is also the reviewer, so they can annotate parts of the code that they may want to rework in the future. As the developer scrolls down the code any issues that were noted in digital ink by the reviewer become easily distinguishable from the code, and also directly related to a specific portion of code. Once again each issue can also contain severity indicators, whereby higher issues would have more importance over issues with a lower priority.

During a peer review it is more effective in terms of code quality and time efficiency to entice the reviewer to simply mark the erroneous parts of the code with digital ink. The reviewer can subsequently let the developer read these annotations so he/she can make the prescribed changes, based on the developer’s higher understanding of the code. As a result during a peer review whether the peer is a friend, mentor, or a tester, they should simply mark possible problems, solving these issues should be left for the developer.

3.6 Lazy Developer

Scenario

I consider lazy developers to be those who don't comment over their code during development, making it extremely difficult and time consuming when someone else has to interpret the code before they can add extra functionality. Commented code is also very important when the code must be maintained in the future, or even tested by another member of staff. When excessive amounts of pages of code are written, and the implementation phase is over, the developer would be reluctant to go back and comment over the entire code. This disregard for commenting code is largely due to the fact that developers don't want to sit at a desk and perform such a tedious and tiresome task. Especially straight after writing out hundreds of lines of code, they must then proceed to re-read their code and write comments that involves even more typing. Therefore having to comment code after the development phase generally results in substandard commenting or even a total neglect for commenting. To entice lazy developers to give acceptable comments after development can be difficult, so a more enjoyable approach is needed whereby commenting can be performed effortlessly that takes place in a relaxing environment away from the office.

Solution

Using digital ink over the coding IDE gives the developer a new commenting perspective separate from the way code is developed and commented. An ink annotation system may even make the concept of commenting code more enjoyable, rather then the dull and tiresome typing approach used throughout the implementation stage. The added enjoyment comes from the digital annotation systems portability and this innovative annotation approach. This capability within a coding IDE allows annotating over code without having to be at a desk. Commenting code can now take place while sitting at the park, on a train, a plane, or even relaxing in bed.

3.7 Requirements

This subsection lists and explains several requirements that satisfy each scenario in order of priority. These requirements are then presented in figure 3.1 illustrating requirements corresponding to each scenario. Once we have an idea of all the requirements, we can select a core set of necessities which can be used to evaluate an IDE ink annotation prototype.

Digital Ink

Due to most development projects involving multiple source files, the tool must handle digital ink over each of these files. This tool must efficiently switch between individual ink panels whenever the user switches between code files. The ink must be free-form, meaning ink can be written anywhere on the digital document without any limitations (Schilit et al., 1998), much like annotating with traditional pen and paper.

Extending IDE

The code review tool must be an extension to an existing IDE. This is so the tool can be used to annotate issues directly over code, while still having all the existing functionality of the IDE available when reviewing code such as: compiling and run, code editing and most importantly the IDE's class and variable navigation features.

Ink Editing

As digital ink annotation provides a computational approach, we can make it easier for the users to edit (move and delete) their existing ink annotations, it is also possible to give users a vast range of colours and pen types.

Preserving Ink

Saving/Loading of the ink annotations is essential for all scenarios. When a code file is loaded, and there are corresponding annotations existing for this file, then these annotations must also be loaded into the IDE. Once a code file is saved, then so too should its corresponding ink annotations.

Ink Reflow

Ink reflow is important for an ink annotation tool due to the active nature of its underlying program code. When the code is altered all existing ink must be repositioned to keep correct alignment with the underlying piece of code.

Severity Indicator

Code review is essentially locating and indicating issues within code and then resolving those issues. Because of this fact the reviewer must be capable of indicating the seriousness of an issue, such that the developer can resolve these issues based on priority.

Annotation Author

When a group of people are reviewing the same document (i.e. during public review, or group review session), it may be desirable to have each author register before they can annotate over a public document. Once registered, every annotation will include the annotator's unique signature, others can then approach them if necessary, for any clarification.

Issue Report

Generating a report at the end of a code review is another practical feature, giving the developer a quick and clear overview of all issues that were noted and need resolving, without the need to open the code project. The report would contain the issues line number, severity, error type, annotation creator and the corresponding ink strokes.

Search Annotations

Being able to iterate through each ink annotation within a code file is an efficient and handy approach to locate individual issues. Searching for annotations can be performed sequentially, or based on their severity, their issue-type or even on the annotations author. Searching a coding project not only involves multiple pages of code, but also multiple files of code, thus the annotation tool should allow the user to search for annotations within either the current code document or the entire project.

Timer

In some situations, having a timer indicating the time spent annotating code is useful. This is particularly a good feature when the developer or project manager asks others to review code, as it gives good indication of how long the reviewer actually spent annotating the code. Johnson (1994) suggested a more effective approach to ensure an accurate review time, and involves counting only each minute where the user interacts with the code (e.g. scrolling or annotating of code).

Merging Annotations

When several people are conducting reviews on multiple copies of the same the file (e.g. during private reviews, or peer reviews done by several peers) all annotations from separate reviewers may need to be merged. This allows all annotations from each reviewer to be consolidated into a unique list of issues, which is then imported into the one code version. The benefits of merging documents annotations are obvious, rather than individually looking at all annotations made by each reviewer on a separate copy of the source code, the developer can have the luxury of all comments in one source file with duplicates removed or hidden. When annotations are merged the annotations author will also show, so any questions can be clarified by its author.

Marking Schedule

This tool can be used for a marker to annotate and give a mark directly over student assignments. This capability to import a group of assignments and a marking schedule into the tool gives similar functionality to Plimmer & Mason (2006). However, as this tool is directly integrated into the IDE it allows all existing functionality that the IDE has to offer.

Scenarios

Requirements

	Teaching	Marking	FTR	Peer Reviews	Commenting
Digital Ink	✓	✓	✓	✓	✓
Extending the IDE	✓	✓	✓	✓	✓
Ink Editing	✓	✓	✓	✓	✓
Preserving Ink	✓	✓	✓	✓	✓
Ink Reflow	✓		✓	✓	✓
Severity Indicator		✓	✓	✓	
Annotation Author		✓	✓	✓	
Issue Report		✓	✓	✓	
Searching Annotations		✓	✓	✓	
Timer			✓	✓	✓
Merging Ink			✓	✓	
Marking Schedule		✓			

Figure 3.1: Individual Requirements for each Scenario

3.8 SUMMARY

This tool will provide the user with an ink annotation code review system directly within the IDE, giving benefits in five distinct and diverse review scenarios. This was described by presenting the previous approach for conducting these specific types of reviews, followed by proposing an approach with ink annotation support. A prioritized list of requirements was then presented, also indicated are the scenarios that would benefit from each requirement.

This tool uses digital ink annotations to represent coding issues, and has the possibility to add additional features to the tool that provides extra assistance to users. Within an academic environment, lecturers would use this tool to teach code to students, and academic markers can use this tool to review and mark students programming assignments. In an industry-based development company, this tool can be used in conjunction with existing formal technical review procedures. General purpose developers can also use this tool to conduct self/peer reviews and also to give a different commenting perspective for programmers. In order to develop a simple IDE annotation tool prototype, we design and implement only a core set of requirements extracted from the requirements above.

Chapter 4

Design

4.1 Introduction

To evaluate the concept of an ink annotation tool within an IDE we formulated a set of requirements from the previous chapter that satisfy many review scenarios. However before we implement these requirements, we need to determine whether or not our tool will be successful. So we first consider a core set of functionality for this ink annotation code review tool. Once we have a working base prototype that can be evaluated, then we can incorporate additional scenario-specific features.

This chapter presents an outline of the tools core requirements. The tool must reside within an IDE and support digital ink strokes that reflow as the underlying code changes. Therefore each ink stroke must be grouped into logical annotation sets then attached to a portion of code, so as the underlying code moves the attached annotations should also move alongside the code. These issues must also be easily modifiable and include a corresponding severity indicator to prioritize the order issues are dealt with. Finally this tool must work seamlessly within the IDE and thus still allows the user to make full use of the frameworks existing functionality.

4.2 Extending The IDE

Previously, IDE's did not support code review and would be reviewed either within a separate application or by feedback on code print-outs. The main problem with these approaches is that source code is non-linear, meaning it cannot be simply printed out then read like book. Due to the non-linearity of code, the reviewer has to jump around to different locations either in the current code file or other code files within the project simply to locate methods or class definitions.

Another problem with reviewing code in a separate application is that debugging and running the project cannot be done within the same program as the reviewing procedure, so a lot of switching between reviewing devices (switching between the IDE and either a separate code review application or coded printouts) is required.

Having an ink annotation tool residing directly within IDE eliminates most of this extra effort and overhead, as source code can be reviewed within the same application that was used for developing the program. Using the code annotation tool as an extension of the IDE allows the user to perform a code review and also allows all existing IDE functionality to be available while they review the code. For example, the code can be compiled and executed which is useful for initial testing; the use of the editor's extensive searching capabilities (e.g. search of method declarations and variable references, and searching of keywords) makes it quick and easy to locate a method definition and all its references. What's more when reading code, most environments use a range of colour coding to indicate specific keywords, and a range of text formatting options for how the code is to be displayed (e.g. fonts and font sizes, tabs) along with automatic listings of an object methods and public variables while coding.

To fully utilize the advantages of an ink annotation within an IDE, an appropriate environment must first be chosen. We must carefully consider the characteristics of each IDE nominee before we begin with the implementation. The decided framework must be one that is popular such that it is widely used which will make it easier to evaluate this code review tool. Also we need an IDE that offers extensibility to implement the tool then integrate this as an extra layer of functionality within the IDE. Two of the most widely used IDE frameworks available today are Eclipse (The Eclipse Foundation, 2006) and Visual Studio 2005 (Microsoft Visual Studio, 2005).

Eclipse is a java-based development environment and has the benefit of being easily extendible as it is 'open-source', thus gives developers no limitations when extending the existing environment. Visual Studio 2005 is another popular development framework (InfoWorld, 2006), and supports many popular development languages including Visual Basic, C#, C++, J# as well as web development support. Visual Studio is a widely used IDE within the software development industry (Microsoft PressPass, 2000), and also gives the opportunity to customize and extend the existing framework by having access to the underlying routines and environment variables. However the disadvantage of Visual Studio's is that it isn't open-source software and so only gives selective access to the underlying routines in the IDE. Visual Studio 2005 was chosen to host our ink annotation tool.

4.3 Digital Ink

This tool must support digital ink annotations within all existing code windows in order to record issues. This is similar to the traditional pen and paper annotations whereby markings are written in red pen to make issues look more apparent to all future readers. Annotating with digital ink has the advantage over traditional pen and paper because it uses the power of computation to easily modify and alter existing ink strokes and their properties. Annotating with digital ink is also more effective than representing code issues by typing regular comment constructs (e.g. '// write comment here' for C#) within the IDE. Therefore representing coding issues as typed comments in the IDE would be visually similar to regular code and code comments as they involve an identical font size and style and so would not always be distinguishable. Digital ink comments become more visible to readers when skimming through code as appose to the developer iterating throughout the entire project to find coded comments that represent a severe coding issue.

The digital ink within this tool must be 'free-form', this implies the user can make use of ink annotation anywhere over the underlying text without any inking limitations of shape or content (Golovhinsky & Denoue, 2002), much like traditional pen and paper annotations. However because an IDE contains other windows (file window, debug window, existing toolbars etc.) as well as the code window, the free-form annotations should only be permitted over the code windows and in the future, also form and class designers. This is usually achieved by attaching a transparent ink overlay over each code window. So as the underlying code window scrolls, so too does the transparent overlay, hence giving the illusion the digital ink is directly attached to the underlying code.

Whether you're writing small or large programs, the project often involves multiple source files. Most IDE's accommodate this by providing a separate tabbed window for each code file (Microsoft Visual Studio, 2005; The Eclipse Foundation, 2006), this means the annotation tool must efficiently manage free-flow digital ink over these multiple windows. Because code reviewers switch between different code windows to view different object classes, loading of this ink during the change of code windows must be efficient.

4.4 Reflowing Digital Ink

The word 'reflow' in digital ink annotation terms refers to repositioning ink strokes as the dynamic underlying context changes. If the underlying code changes the ink must also be realigned to remain

consistent with its coded context. If the code document is static and hence the text will never be altered once the inking is finished (e.g. during a marking scenario), then demand for reflow is reduced, but not eliminated. Static documents as explained in Golovchinsky & Denoue (2002) can still incorporate zooming in and out where the change in aspect ratios must also be used to resize the ink. Also certain devices may show documents with different display properties (e.g. different page lengths, different margin widths and different font sizes) and so it is possible for text to wrap around to the next line or even over a new page. Even though no text is added or removed from the document situations still occur where reflow is important.

Because of the dynamic nature of code, where a reviewer would represent issues and the developer would alter the code to resolve these issues, the code will be edited regularly. This will mean reflowing digital ink to realign with the underlying edited code is considered an important feature for this tool. Due to the limited time scope of this project we restricted the reflow capability to handling only vertical changes in text. Code is line-based, where the text will not wrap around to the following line, meaning we are fortunate enough to attach comments to individual lines. When a code line moves as a result to code being added or removed, or if a portion of code is moved to another portion of the document, then the digital ink strokes must also realign to represent the new line location of the code statement. To successfully reflow annotations, there are three important steps (Bargeron & Moscovich, 2003): ink strokes must be grouped into a comment set of ink strokes; this group must then be attached to specific point of reference, in this case a line; finally these comments must move along with the underlying code.

4.4.1 Grouping Ink Strokes

The first step for successful reflow requires grouping individual ink strokes as they are made, then associate them to one specific issue. Once grouped, the annotation issue must then be anchored to a particular piece of code. When grouping ink strokes Bargeron & Moscovich (2003), Brush et al., (2001) and Chiu & Wilcox (1998) suggest the time between each stroke (temporal) and the location of the ink strokes (spatial) should both be taken into consideration.

Temporal grouping refers to an ink stroke being made and the next stroke occurring shortly after (within a certain time span). Then it is assumed to be grouped with the previous stroke, e.g. writing the next character of a word, dotting an ‘i’, or crossing a ‘t’. Golovchinsky & Denoue (2002) suggests the time span between related ink strokes can reach to approximately half a second (500ms). However we cannot rely on temporal data as people may add to existing annotations.

Spatial grouping refers to ink strokes being made within close proximity of an existing ink group, e.g. appending strokes or comments to a previously existing stroke group. In this tool we need to decide how close a stroke is to a current group of ink strokes. When measuring the distance between the recently made stroke and the bounding box of the current group of ink strokes, if this distance is outside two code lines we would consider it a new group. If the annotator wants an annotation to be part of a previously existing issue group then they must first select the annotation group. Selecting this group in this way will mean further ink strokes within close proximity will be grouped with this selected group.

4.4.2 ANCHORING GROUPS

The second step for successful reflow requires each issue to be anchored to a particular word or portion of text. Techniques used for anchoring groups of ink strokes within a text document are not suitable for code. With text it is not appropriate to attach ink to a specific line, this is because text documents generally include margins, and therefore words regularly wrap around to the next line or even the next page. A text document often involves attaching the group of ink strokes to a specific sentence or portion of text (Brush et al., 2001). This type of anchoring works well for text documents because it is uncommon to find two sentences exactly the same, but not code documents. Throughout code there are many instances where code instructions are exactly the same (e.g. loops, return statements and catch statements), so anchor points that correspond to words in a sentence won't always correspond to one unique line of code. This makes it unclear as to what sentence the issue is grouped to when the code is moved or deleted. Code documents have no width margins and so words don't wrap around to new lines or pages, each statement of code resides on one line. This allows us to use line numbers as suitable anchor points for grouping ink strokes.

4.4.3 MOVING GROUPS

Finally, as the underlying text moves, so too should the existing comments. When dealing with dynamically changing documents its underlying code can go through several types of vertical or horizontal changes. There are two possible vertical changes that can occur where the digital ink strokes must move as the code moves. First a line or group of lines can be inserted or removed, in this case all lines and ink strokes that follow the modification must move down or up. Secondly, it is also possible for a group of lines to be moved to another portion of a document (e.g. to reorder method definitions), again

any associated ink strokes must reflow into their new location. Horizontal changes occur when characters or words on a line are altered or moved. However, because a coding IDE is not a text document, each code line exhibits a strict structure of methods and variable names. If the code line was changed significantly, such that it looks different to the original line, then the line would probably exhibit different functionality, therefore the annotations context would be lost and the ink would also lose meaning. However if a line was to be altered slightly, for instance changing variable names or altering the number of parameters in a method call, then its meaning and its structure remains the same. In this case, if the ink annotations are not reflowed they would only be slightly offset from its anchored text. Research shows that ink strokes slightly offset from their original context, do not pose a big problem for users (Bargeron & Moscovich, 2003). Hence during changes on a line, existing ink strokes would either totally lose context and the user should delete them, or if the change was minimal the ink strokes would only be slightly offset to its original context and so in both cases the digital ink need not be reflowed. So we can ignore the need for horizontal ink reflow in this code review tool as its not an essential benefit.

4.5 MODIFYING INK ANNOTATION

One of the biggest problems with traditional pen and paper is the difficultly of modifying existing annotations. Once the marker or reviewer has marked on the script with red ink, the ink strokes are fixed. The strokes can be concealed but this can look messy, ink strokes can never realistically be entirely removed. Annotators could however use a pencil so that their markings can be erased then rewritten in pen once they are happy with their comments. Using computation to mimic traditional pen and paper annotation is a good solution to modifying ink, because it allows the ink to be selected (individually or as a group) then cleanly erased or repositioned.

With the benefits of computation, the tool can use many different colours of ink, types and width of pen tip as well as rulers and erasers, all at a tap of a button. However, in terms of ink properties which would benefit an IDE code review tool, the most useful is the ability to select different ink colours to represent different types of issues. In previous annotation research Marshall (1997) has ascertained that annotators of educational textbooks often use different colours to represent different ideas and meanings. This is also consistent with Heinrich & Lawn (2004) where a markers annotations can be colour categorized. This colour coding approach could also be successfully used within a code review tool to represent different types of issues and defects found by the reviewer, as suggested by Basili et al. (1986) and Myers (1978) where code review is more effective at finding different types of errors. For instance an issue written in blue could indicate a logic error, whereas an issue written in red could represent an arithmetic error.

4.6 ISSUE SEVERITY

When the reviewer annotates code, they signify whether there is an issue in that portion of code. An issue could be a negative statement suggesting a defect, or positive feedback praising a portion of the developer's code. As its common code review practice to indicate a corresponding severity level for an issue (Fagan, 1976; Johnson, 1994), a code review tool should also give the reviewer the capability of setting severities. This severity indicates the importance an issue should be resolved in terms of low, moderate or high. Having said that, the reviewer may want to praise the developer with positive feedback, a positive indicator should also be available.

Once the developer receives the reviewed version of their project there could be possible time constraints or deadlines that need to be upheld. So rather than resolve each issue one-by-one in a sequential order, it may be more valuable to prioritize issues, to ensure severe issues are resolved first. For example, consider a demonstration of an online banking website presented to a bank manager at the end of week. It would be more beneficial to spend time resolving dangerous security and math logic matters, rather than spend time adjusting the low priority requirements such as renaming variables and procedure instances. Once the developer has successfully handled an issue, they should be able to adjust the severity level to indicate to the reviewer (for future reviews) that the issue has been resolved.

Indicating an issues severity can also be effective during other review scenarios. For instance, severities would be useful for assignment markers in an educational setting, where the severity level of a defect correlates to the amount of marks deducted. The marker could also use a positive severity to indicate encouraging remarks for certain portions of code. Severities can also be used in a teaching environment, as they can represent the importance a specific portion of code has on an upcoming assignment or exam.

4.7 PRESERVING INK ANNOTATION

The preserving of digital ink is an important function that any annotation tool must support. Preserving refers to saving the current ink strokes to non-volatile storage (disk) once the user has finished annotating, then once again loading the ink strokes when a user wishes to continue annotating. This support for seamless saving and loading of ink, also involves creating, locating and integrating the digital ink into a project.

Once the reviewer has a copy of the code to review it must be easy for them to create an ink file so that they can comment over the underlying source code. Once the code has been annotated the digital ink must then be saved just as the source file is saved, and stored in a location related to the project.

However because code review can be done by someone other than the developer, the created annotations must be sent back to the developer to resolve the reviewers feedback. The digital ink must be easily transferred and then integrated into the developer's project. Because the developer has the original copy of the source code and it's the reviewer's job to simply raise issues and not resolve them, all that needs to be sent to the developer are the ink files, not the entire project. If the code reviewer makes changes to a source file, then a copy of that file must also be sent back to the developer or possibly risk consistency problems.

Once the developer has the copies of the ink files, it must be easy to integrate them into their copy of the project for viewing. So when the developer opens the project, an automatic search for any corresponding ink files is made, if any ink exists it is loaded along with its corresponding source code.

4.8 SUMMARY

This chapter presented an overview of the core design requirements for this IDE ink annotation tool. The design covers both essential and high priority functionality from the previous section. For now we feel these design features are enough to evaluate the tool. If the concept is successful, then this design could be extended to provide more scenario specific features from the previous section. The next chapter uses these requirements to discuss both the implementation and design refinements made during development of this working prototype.

Chapter 5

Implementation

5.1 INTRODUCTION

It was decided the 'Rich Code Annotation' (RCA) tool would be developed using Visual Studio .NET 2005 IDE as a plug-in for two reasons, one its vast range of different languages (including C++, C# and VB, which are all taught at this university) and two the framework's exceptional popularity. Within this university's computer science department includes courses that use several of these .NET languages and with its popularity within industry, it gives us greater opportunity for both academic and industry evaluations and tool exposure in the near future.

The RCA tool was designed and implemented for use on tablet PCs, where digital ink annotations can be made directly onto the single input tablet display using a stylus. This then gives a pen and paper like appearance and also includes certain benefits inherited from the traditional pen and paper approach. This tool can not only be used on tablet PC's, but it is also useable on large desktop computers with the aid of a tablet USB input pad. However when operating any ink annotation tool on a desktop computer the annotating will often be considered a little more difficult, as the user is not dealing directly with the screen. Additionally its size and weight eliminates any portability advantages that are seen with tablet and other smaller computers.

The RCA is implemented in C# as an 'extensibility' project that combines both the Visual Studio's DTE (Development Tool Extensibility) automation object and the Microsoft Ink API. The automation model gives access to Visual Studio specific properties, methods and events from the current instance of the IDE framework. The range of property information available to the developer includes information from the options menu found within the "Tools" tab (e.g. 'Environment' and 'Text Editor' settings etc.), information relating to both the active window and other existing windows (such as their name, type and the content within the window), and lastly a solution property that gives information concerning all open projects and their locations within the current environment. The variety of IDE event notifications available include the text editor's line changed event, the document's opened and saved events, and also window created and activated events.

The Ink API allows for collecting and grouping ink strokes, selecting and moving ink strokes and manipulating the digital ink properties and can be found in Gocinski (2004) and Jarrett & Su (2002). There are also several events that fire when digital ink is added, deleted, moved or selected, these events are all essential for this tool.

This chapter starts with an overview of the RCA tool by giving a high level description, followed by a more comprehensive implementation description of each individual component of the RCA. These components include the ink toolbar, extending the IDE to create an ink window; linking annotations to a specific code line; grouping ink into an annotation; reflowing ink as the underlying code is modified; editing ink; signifying issue severity; synchronizing the code window with the ink window; and finally saving and loading ink and maintaining consistency. We also describe the implementation of several modifications that were realized from our informal usability testing.

Before we continue, a brief list of terms is required, which is then given more weight in figure 5.1. An ink 'stroke' refers to the ink that is generated while the pen is touching the screen to the time when the pen lifts off the screen. A 'comment' refers to a group of ink strokes used to make a descriptive comment. A 'linker' refers to the ink stroke that is used to link the comment to a particular portion of code. This is distinguishable by the broader ink stroke. Finally, an 'annotation' refers to the complete set of strokes that is used to generate the issue (a linker and its comment).

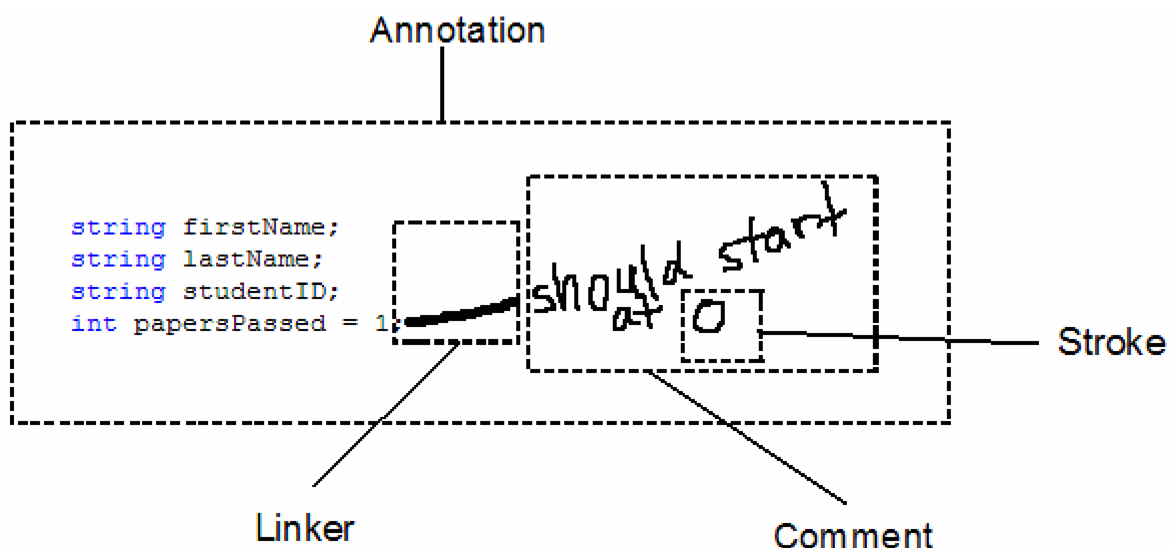


Figure 5.1: An Annotation Example

5.2 OVERVIEW

Our final version of the code annotation tool was implemented as a separate 'Ink Window' within the IDE (figure 5.2). Anytime the IDE loads in a code window a search for its corresponding ink file is performed, if found the IDE will also load this ink file and display it as an 'Ink Window'. If a user brings up the code file, and clicks on the pen button from the 'Ink Toolbar', the code windows corresponding ink window is activated (made viewable). If no ink window exists for that code window then a new ink file is created and displayed as a new 'Ink Window' based on that code window.

When a user wishes to review code it must first bring up the 'Ink Window' for that code, they can then raise an issue by first generating a link to the specific piece of code, followed by free-form comments describing the issue. The link between the code and the commented issue is called a 'linker stroke', where the user must explicitly select the portion of code that corresponds to the commented issue. To create an annotation, a linker must first be created, it can be represented as either a line stroke alongside a single line of code or as a circle stroke that encapsulates several lines of code. Once the linker is generated any further nearby ink strokes made shortly after will be grouped as a comment and attached to the linker, this is considered a single annotation. Basically, an annotation describing a code issue will always include one linker and a group of zero or more ink strokes commenting on the linked portion of code. An annotation is initially given a moderate (orange) severity that is attached to a particular code line (based on the linker's location) which is then adjustable in such a way where the user can select a desired rating. This severity is also used to indicate the line that the annotation is attached to. So as code on this line moves to a new line, both the severity indicator and the annotation moves with it.

With the RCA tool, the user edits code in the source file's code window and can represent a coding issue by annotating over the source codes related ink window. This ink window is saved as an ink file, and if the code within the code window is saved its related ink window (if exists) is also saved. When annotating the user can erase, select and move strokes either individually or as a group, the user can also re-colour strokes by selecting the annotation and selecting from a vast range of ink colours. The sections that follow give a more detailed description of how each component was implemented.

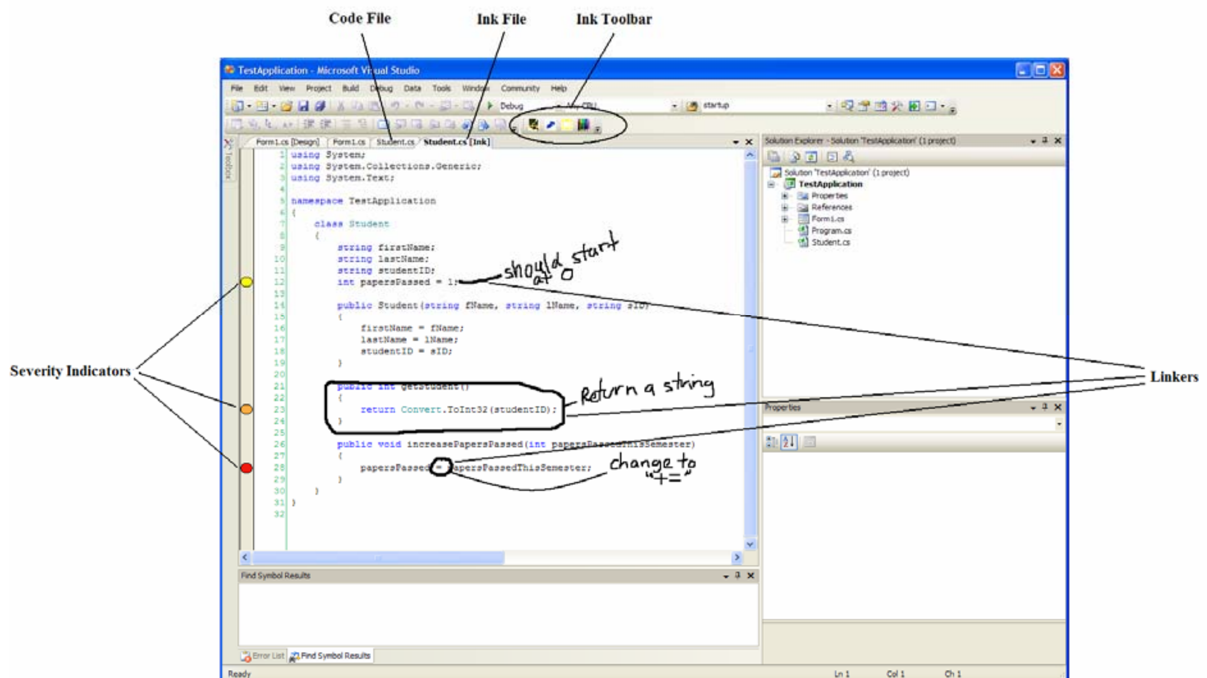


Figure 5.2: The RCA User Interface

5.3 INK TOOLBAR

The ink toolbar for the RCA tool is shown in figure 5.3, and consists of four buttons; these include three annotation modes (pen, eraser and selection) and one ink colour menu:

- pens
- eraser
- selector
- colour dialog

When the pen button is clicked while viewing a code file, its corresponding ink file will be created (if non existent) then activated making it viewable to the user, who will then be able to directly make annotations over the ink window. If the pen button is clicked while already viewing the ink window it deselects any previously selected button and selects the pen. This selection changes the previously selected annotation mode (eraser or selection mode) and allows the user to create digital ink annotations directly over the viewable ink window.

While the user is annotating over the ink window the eraser button is available and used to delete ink strokes. The select button gives the user an option of selecting an individual ink stroke or a group of strokes for modification, once a stroke or strokes are selected they can be moved, re-coloured or deleted.

When selecting strokes, either individual strokes from an annotation can be selected, or if an annotations linker is selected it automatically selects all ink strokes within the same annotation group. Therefore any moving, re-colouring or deletions that occur on a selected linker stroke will also affect all strokes in the linker's annotation group. For instance, if the link between the code and the comment is deleted, then the linker stroke and all other strokes within the same annotation group are also deleted.

The colour dialog button gives users the opportunity to alter the colour of their annotations. Marshall (1997) suggests that many academic annotators may want to colour-code their comments, and this is also consistent in Heinrich & Lawn (2004) that suggest markers may want to categorize their feedback. Code reviews are valuable for discovering different classes of issues (Basili et al., 1986; Myers, 1978), so colour coding within a code review tool can be used to clearly represent these error-types. The RCA tool allows colour coding individual annotations to represent different issue types, this is achieved by using the colour dialog menu. The change in colour affects both currently selected strokes and future strokes. If the linker is included in the selection, then once again all the strokes within the same annotation group will also be re-coloured.

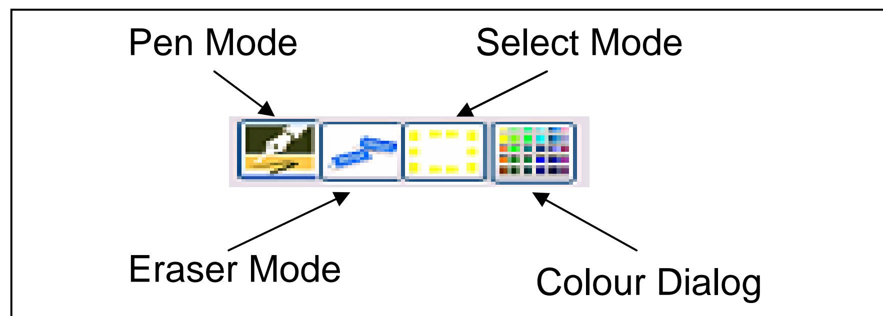


Figure 5.3: Ink Toolbar

5.4 INK WINDOW EXTENSION

We investigated three possible approaches to support ink over the code windows. Initially we wanted to add a transparent overlay to each code window, such that the digital ink would be visible while annotating and coding. This would definitely be the best approach, however due to the limited event notifications within the Visual Studio framework, we encountered our first problem. When the code window is scrolled we need an event to fire, so we can also scroll the transparent overlay by the same amount, to ensure the digital ink on the overlay would still be correctly aligned with the underlying code. However no event exists in the Visual Studio automation model that would notify us that this code window is being scrolled. Therefore we have no way of knowing when to realign the digital ink in the overlay with its underlying code. Another approach was needed.

The second approach was an attempt to create a new ink window that was tightly-coupled to its code window, much like the coupling between the code view of a windows form and its associated 'design' view in a Visual Studio application. The expectation with this idea was that we could create a tool window resembling the code window that included a rich text box containing a reference copy of code from the code window, thus having a direct link between the two windows. With this approach we thought it would be possible to specify the link within the tool windows properties, similar to how a windows application code window and its related design window are linked. In this case when code is added within the code window, we wanted the code within the related ink window to automatically update its contents to keep consistency with the code window. This approach was also found to be far too complex to implement with limited middleman access of the Visual Studio framework. Both our first and second approaches may have been more successful within an IDE such as Eclipse due to its open-source structure.

Finally we took the idea of the second approach to create an ink window that would be logically related to the code window, this meant whenever the code in the code window changes, we explicitly update the code within the ink window. This was the approach we took for our final implementation of this tool. The ink window was created first as a distinct tool window hosted by an activeX component, much like the 'Shim Control' example developed by Conwell (2004). The tool window contains a user control that resembles the layout of the code window, it includes a rich text box to hold a copy of the code, line number and break-point margins as well as horizontal and vertical scroll bars, the overall ink window is then attached as a 'tabbed document' in the project.

There are three main drawbacks with the final approach. The most prominent drawback is that the ink annotations are not visible when editing code. This problem can be reduced however, by using a horizontal separator where the code window can reside on one side of the separator and the ink window can reside on the other. The second drawback with a separate ink window is that the ink file is considered separate from the projects solution, and so it doesn't appear in the solution explorer. This can be resolved in the future by adding a dummy file to the project with the same name as the ink window. Such that when this dummy file is selected, rather than make this dummy file appear we make the actual ink window appear, giving the illusion the ink file is part of the solution. The third problem is that the ink window is physically unrelated to the code window and so the ink window must be explicitly refreshed by the tool whenever the coded content changes. This requires the rich text box and its content, the line number and break-point margins as well as the lengths of the scroll bars to be updated every time the ink windows code content needs refreshing.

5.5 LINKER

When the user locates an issue or defect within the source code, they represent it via free-form digital ink comments within close proximity of the appropriate piece of code. The comments relating to this piece of code must be attached to a single line number, so when the code statement on that line number moves up or down to a new line number its associated digital ink annotation can follow. We use this 'linker' concept to link a single code-statement or several code-statements to an annotated comment, and this comment is attached to a specific line number based on the type and vertical position of this linker stroke. This linker allows the user to efficiently establish the exact portion of code that relates to a specific digital ink annotation.

The tool incorporates only line linkers and circle linkers because they are part of a common group of annotations as explained in Bargeron & Moscovich (2003) and O'Hara & Sellen (1997). Highlighting and underlining are not very appropriate for code, based on the notion that they are generally used as skimming devices to "keep the readers attention" (Marshall, 1997) and is not effective for shared annotations as they include only a text-anchor with no comment (Marshall & Brush, 2004). Marginalia involves an anchor to a portion of text followed by a comment in the margin and is therefore more appropriate as it restricts code reviewers to generate shared annotations that are easy to interpret by others. A line linker is used to link a single code statement to an annotation, and the circle linker is used to encapsulate a several code statements that can be linked to a comment.

It is common for ink annotation tools to use two separate modes one for inking, the other for recognizing gestures (Lin et al., 2000; Moran et al., 1997). Li et al. (2005) investigated different approaches for switching between these modes. Our tool uses a similar approach with one main difference, the switch between mode changes is performed automatically by the tool rather than by the user. Initially the mode is in 'linker-mode', which is indicated by a hand symbol as the mouse/pen cursor, a stroke in this mode is used to create the linker stroke. Once this stroke is made there is an automatic mode change from 'linker mode' to 'inking mode', this is indicated by a change in mouse/pen cursor from the hand symbol to a pen symbol. In this mode all nearby strokes are assumed to be part of the same annotation group as the linker stroke. After three seconds of no ink activity the mode will change back to 'linker mode' where the user can make another annotation for a new issue. As mentioned in the previous section, the user doesn't need to wait the full three seconds to generate a new issue. If an ink stroke is made a few lines away from the current annotation, the tool assumes the ink stroke isn't related to the current annotation group, and therefore will be considered a linker stroke for a new annotation group. Whether the user is in 'linking mode' or 'inking mode' they can add additional strokes to an existing annotation group. They simply select the annotation by tapping the pen directly over one of its ink strokes, this sets the current annotation mode to 'inking mode' and the new ink strokes will be grouped with this selected annotation. The linker is essentially the first stroke of an annotation and distinguishes itself from regular ink strokes as it made with a thick felt-tip stroke.

When in 'linker mode', a linker stroke is considered a line linker if it is not recognized as a circle. A line linker is simply a line with a start and an end point and as annotators commonly write left to right, the line linker is attached to the code-line closest to the start point of the linker, as shown in figure 5.4. It is expected annotators will attach their comments near the end point of the linker as there is limited room in the left-hand margin and their is usually more room on the right-hand side of the code.

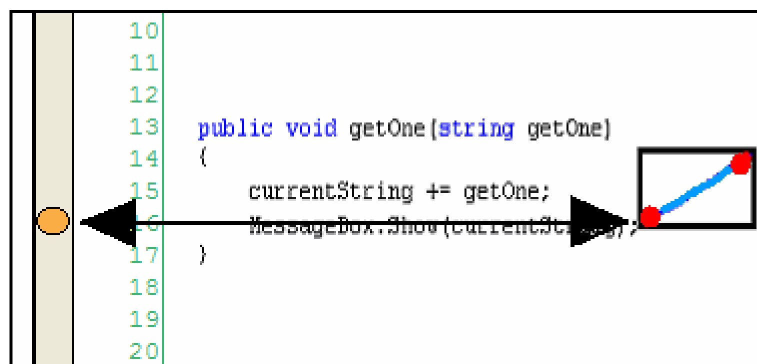


Figure 5.4: Line Linker

If the linker is recognized as a circle then this implies the start and end points of the linker are sufficiently close to one another such that they complete or almost-complete the circle. A circle is formally recognized in this system by having the distance between the first point and last point less than the hypotenuses of 25 percent of the bounding box's width and height. That is, a linker would be considered a circle when the length ab is smaller than the length $a'b'$, as shown in figure 5.5. If the linker is recognized as a circle then the corresponding annotation is attached to the line closest to the vertical mid-point of the circle's bounding box.

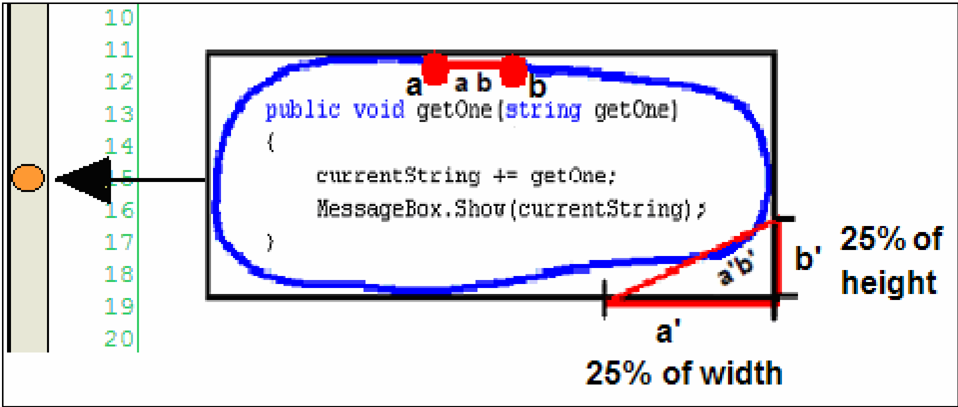


Figure 5.5: Recognized Circle Linker

From our usability tests a common problem was encountered. The users frequently made the mistake of being in 'inking mode' when they were actually in 'linker mode', so it was realized the tool needed to provide a better indication of the current annotation mode. This problem arose due to an ink stroke being made shortly after the change in annotation mode and the mouse pointer was only a subtle indication of this change. The result of this problem requires annotators to delete this ink stroke, and then reselect the previous annotation group to append more ink strokes.

To represent this mode change in a more obvious manner, the tool was adjusted by encapsulating the current annotation group with a red bounding box, as shown in figure 5.6. This problem is now removed as the users have a better indication of the annotation mode, as well as showing them the current annotation that a new ink stroke will be grouped with. This modification is also useful when there are several annotation groups in a close proximity. This indicates any strokes that are not entirely encapsulated in this bounding box as being within a different annotation group.

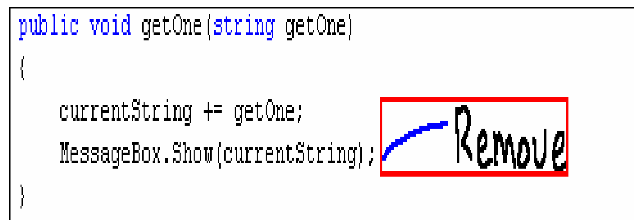


Figure 5.6: Annotation Encapsulated within a Red Bounding Box

5.6 INK GROUPING

With the linker stroke we have attached to a particular portion of code, we can group any extra comment strokes made in ‘inking mode’ to an annotation group. Grouping the ink with its linker stroke allows the system to correctly realign the digital ink to the dramatically changing code context. This adjustment of digital ink is referred to as ‘reflowing digital ink’ and is described in the next section. First all ink strokes must be grouped in annotation sets, such that the tool can move the entire annotation as a group when its underlying text is changed. Ink strokes are commonly grouped based on two characteristics, the time between the current stroke and last stroke, and the distance between the current stroke and the last stroke (Bargeron & Moscovich, 2003; Brush et al., 2001; Chiu & Wilcox, 1998).

Temporal grouping implies two strokes were made within a short time period of one another (a short time between the end of one stroke and the start of the next stroke) and thus are considered to be part of the same comment. Consider when the user generates the characters ‘fix’, once the user writes the ‘f’ stroke the ‘i’ stroke is made shortly after, therefore should be considered part of the same annotation group as ‘f’. Temporal grouping also relates to multiple words, for example the sentence ‘fix logic error’, once the annotator writes ‘fix’ because the word ‘logic’ is written shortly after, it too is considered part of the same annotation group. Golovchinsky & Denoue (2002) has shown the average time between ink strokes for the same comment group is approximately 250ms and the average time between words that are part of the same comment is approximately 500ms (half a second). As soon as an issue’s linker is written and the tool is set from ‘linker mode’ to ‘inking mode’, this system allows three seconds between annotation strokes before switching back to ‘linker mode’. This extended time takes into account the user’s thinking process between multiple words. However the user doesn’t have to wait the full duration before generating another annotation, as this tool also takes into account the location of ink strokes.

Spatial strokes made near existing strokes should be considered part of the same annotation group, for instance going back to an existing annotation group to dot an 'i' or cross a 't', adjust the neatness of a character, or even add a few extra words. Within this review tool the developer may wish to append the word "Fixed" after each resolved issue, so when the code is reviewed again the reviewer can easily see all previous issues that have been resolved by the developer. Unfortunately this property is more difficult to implement due to the algorithm having to make an intelligent decision. The question is whether the stroke should be grouped to the annotation group one line above, or should it be considered a new separate issue. These ambiguities mean we considered a slightly different and more accurate approach to this spatial characteristic.

We consider two spatial characteristics within this tool, we want to 1) make it easy and accurate for users to go back to an existing comment and add additional strokes to that annotation group, and 2) we also must acknowledge that an ink stroke made a reasonable distance away from the existing group of annotations should be considered an ink stroke for a separate annotation group.

We implemented these two spatial ideas as follows:

- If the user is in 'linker mode', they can activate an existing annotation group by tapping any stroke within this annotation group, new strokes made nearby are then stored and added to the selected annotation. This makes appending ink strokes to an existing annotation group easier and more accurate.

- If the user is in 'inking mode' and a stroke is made outside the annotation region, then even though we are in 'inking mode' the stroke is considered a linker for a new annotation group. As the ink stroke was made a significant distance away from the current annotation it is therefore represented as a new annotation. This doesn't require the annotator to wait the three seconds before generating a new annotation group.

Another problem encountered from our usability tests involved the spatial constraints when grouping ink. Creating an ink stroke a few lines below the current annotation seemed ambiguous to users, as they were unsure if the stroke would be grouped with the existing annotation group or as a linker stroke for a new annotation. This problem generally occurred when users ran out of annotating room on the right and wanted to continue writing underneath to the next line, or as they annotate a line, and decide another issue should be annotated a few lines below. The users wanted to know for certain before a stroke is made if the next annotation would be grouped with the current annotation, or if it would be considered a new linker stroke.

To solve this spatial ambiguity problem, we considered the fact that annotators normally write from left to right, then down to the next line. Based on this assumption we reshaped our newly created bounding box by giving it extra height and width below and to the right, as shown in figure 5.7.

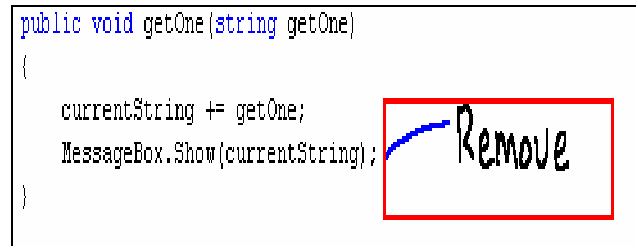


Figure 5.7: Annotation within an Extended Bounding Box

If any point of the new stroke is within the spare space of this bounding box, then that stroke is grouped with the current annotation group. If no point of the stroke is made inside this bounding box, then it should be considered of substantial enough distance away to assume the ink stroke should be interpreted as a linker for a new annotation group. When an ink stroke is added to an annotation, then a check is needed to determine if the red bounding box needs reshaping. After several different approaches it was decided the bounding box should not be adjusted every time an ink stroke is added to the annotation, as it can be visually disrupting to the user. Therefore reshaping should only occur if the newly added stroke leaves little room to continue annotating. So rather than extending the bounding box's size after every ink stroke is added, the system extends the box only if it's necessary.

5.7 Ink Reflow

Probably the most difficult and widely discussed feature in ink annotation research is reflowing digital ink when the underlying document is active (Bargeron & Moscovich, 2003; Brush et al., 2001; Golovchinsky & Denoue, 2002; Phelps & Wilensky, 2000). When incorporating ink annotation within text editing tools there is a high demand for reflowing ink strokes. When the text changes in a document containing ink strokes, the location of the digital ink must continue to remain consistent with its underlying attached context or risk the user reverting to traditional annotation.

Because an IDE is essentially a text editing tool, once the document has been reviewed and the reviewed code is sent back to the developer, the developer would almost certainly adjust the code to resolve the annotations. This may result in lines having to be added or removed within the code project. All subsequent ink annotations in the later part of the document must therefore realign themselves, such that the ink's meaning is still consistent with the underlying code.

We handle ink reflow in this tool by capturing the IDE's 'LineChanged' event that fires whenever a line of text in the code window is changed. However this tool involves added complexity because it is possible to add and remove a group of code lines. We use the difference between the number of lines before and after the line changed event to ascertain the number of lines that have been added or deleted. This means the tool must handle four reflow situations.

- If no lines are added or removed then we don't need to reflow any existing ink strokes.
- If x lines are added at line y, then all annotations below line y must move down x lines.
- If x lines is deleted ending at line y, then all annotations below line y must move up x lines.
- Any annotation strokes that exist within deleted lines must also be removed.

It is also important to consider the start and end points of the keyboard cursor within the range of text during a deletion, for instance in figure 5.8a the text range starts at the start of the line and ends at the end of the line, meaning we include both the first and the last line as being deleted. However in figure 5.8b the start point of the text range doesn't include all of the first line and so the first line can not be included in the deletion, likewise the last line can not be included in the deletion due to the entire line not being within the text range.

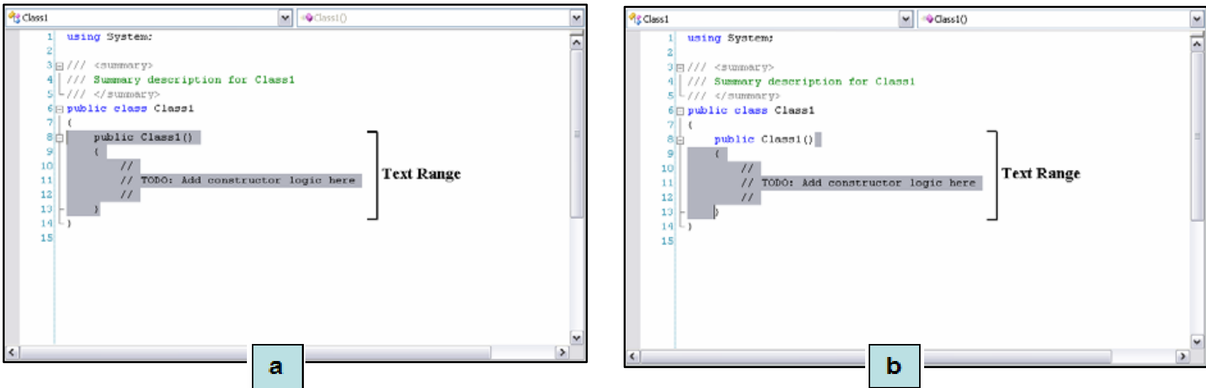


Figure 5.8: Deleting Text Ranges (a) Including the First and Last Line, (b) Excluding the First and Last Line

5.8 Ink Editing

Ink editing is an important part of any ink annotation tool; it offers the user an increase in annotation flexibility by supplying them with functions not readily available with the traditional pen and paper approach. One advantage of digital ink is the ease of modifying and concealing existing annotations, and having a variety of inking instruments (rulers, ball-point and felt tip pens, and a range of colours) available at the tap of a button.

5.8.1 Moving

The user may want to move the annotation to a new location. We support this moving functionality by allowing the user to first select the appropriate linker (using the selector button from the ink toolbar), this will include all the strokes associated with the linker which can then be moved as a group by dragging the group of the strokes. Once the annotation group is moved, the issue along with its severity indicator is reattached to its new line, based on the new location of the linker.

5.8.2 Erasing

Ink strokes can be erased either individually or as a group by using the eraser option from the ink annotation toolbar. If the code reviewer wants to remove some strokes in order to rewrite a comment, then they simply select the eraser then tap/drag over the strokes they want to remove.

A reviewer may want to remove the annotation as a whole. This can be done by simply deleting the linker stroke of an annotation. If the eraser is used on a linker, a warning is made to the reviewer confirming their decision to remove the entire annotation, as removing a linker also removes all its corresponding strokes within the annotation group.

5.8.3 Ink Properties

With traditional pen and paper annotation, research found that students and markers tend to use some sort of colour coding system, where different types of annotations are indicated using a specific colour (Hienrich & Lawn, 2004; Marshall, 1997). Because this tool deals with code, there are many types of problem categories a code issue could fall into, for example an issue could be based on logic, code inefficiency, security flaw, inaccurate loop control or a violation of a company's coding standards (Humphrey, 1989). A tool of this kind may benefit from allowing reviewers to associate a distinct colour for a specific type of issue. When the reviewed code is returned, the developer would have a better idea of the types of errors they are more prone to make, and perhaps this would help them reduce these errors in the future (Fagan, 1976).

This tool allows the user to change the currently selected colour of the ink strokes at any time by choosing an appropriate colour from the colour dialog button in the toolbar. It is possible to select the entire issue (by selecting the linker) or select several strokes within an annotation, if a new colour is chosen it will affect the currently selected strokes, and set the current colour of any future strokes. However if no strokes are selected then the new colour will be used as the reviewers current colour and all future ink strokes will be in this new colour.

5.9 Issue Severity

Issue severity indicators can be used throughout the code review procedure, for instance during the public and private review stages, and also within the review inspection report that has been generated from the group review meeting. Signalling the severity of an issue is not a necessity for this tool, however it does give the reviewer an opportunity to advise the developer of what issues may need the most attention (Fagan, 1976).

The severity of an issue is automatically attached to the line that its annotation is attached to, as shown in figure 5.9a. This indicator is represented in this tool using a dot symbol in the grey left-hand margin bar of the ink window, with similar representation of a break point symbol in the code window. The severity indicator for an issue is initially set to the colour orange, and alternates between green, yellow, orange and red, and are adjusted by tapping the indicator like a button. A green coloured indicator can be used by the

developer to signal (for future reviews) that a specific issue has been resolved. This indicator could also be used by the reviewer to give the developer positive feedback on their code (especially useful during a marking scenario). The other indicators can be considered to represent low, medium and high severities respectively.

When looking through the ink annotations in the ink window, the developer would have to switch to the code window to rewrite the code in order to resolve these annotations. Because the ink strokes are only visible in the ink window and not within the code window the developer would need a point of reference indicating the location of the annotation in the code window. So when an annotation and its severity are attached to a line within the ink window, a bookmark symbol (as shown in the grey margin of figure 5.9b) is attached to the same line in its corresponding code window. A bookmark can be set by the user manually, but can also be set by the RCA to indicate an issue was raised on this line in its related ink window. This approach allows the user to keep track of the whereabouts of the annotation in the ink window when resolving them in the coding window.

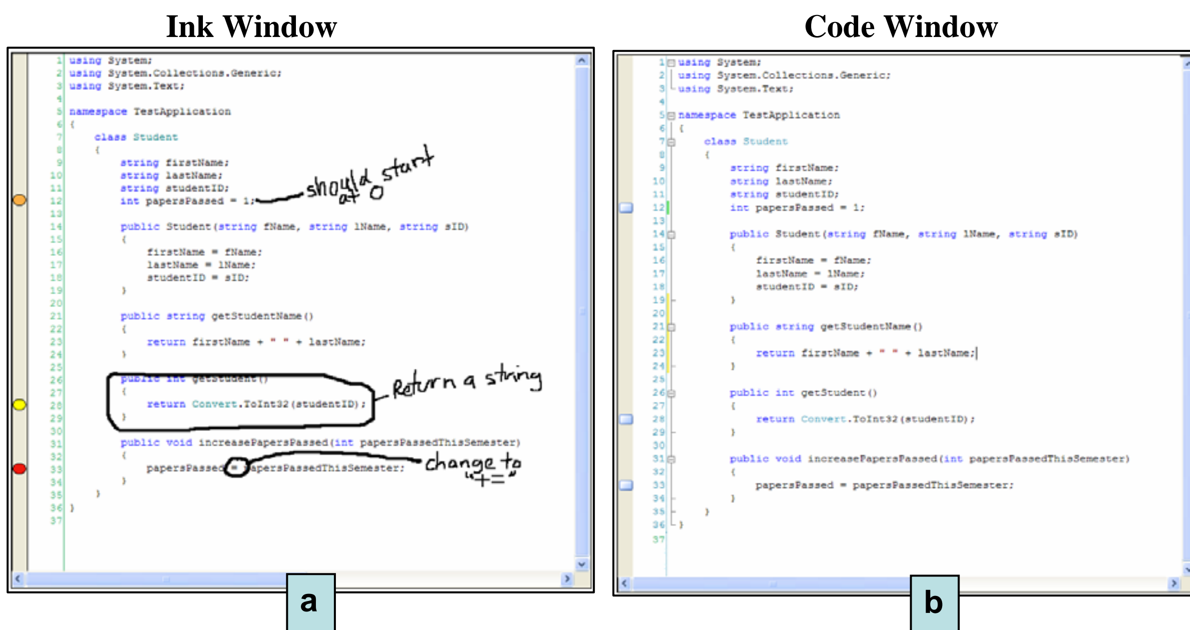


Figure 5.9: Indicating Severities includes:
(a) Severity Indicators shown in the Ink Window
(b) Bookmarks in the Code Window Indicating Annotations Exist in the Ink Window

5.10 SYNCHRONIZING WITH THE INK WINDOW

When moving between the ink window and the code window we must synchronize the windows so they both show the same portion of code. When the user scrolls down to the next annotated issue within the ink window, they switch to its corresponding code window to edit the code and resolve this annotation.

However, before the user can resolve the annotation they must also scroll through the code window to locate the same position of the annotation as seen in the ink window. The bookmark symbol in the margin makes the annotation easier to find, but the problem is that the code window will also need scrolling. This switching of windows must be performed with as little disruption as possible. Having to also scroll to the appropriate piece of code in the code window could be time consuming and risks breaking the developer's concentration.

We decided to improve this efficiency by synchronizing the code window with its corresponding ink window. When viewing an issue in the ink window, if the user then views the corresponding code window, the tool would automatically set the viewing area of the code window, such that the same portion of code seen in the ink window is also viewable within the code window.

This was implemented by intercepting the window activated event, and checking the current active window with the previously active window. For instance if the current window is a code window and the previous window was its corresponding ink window, then we set the new location for the keyboard cursor to the same line as the top-most line displayed in its corresponding ink window. So the top-most line in the ink window will also be the top most line in the code window. Now the reviewer simply scrolls to the next annotation in the ink window. When its time to switch to the code window to resolve this annotation, they won't be disrupted from their task as the code of interest is located in the same viewing area.

5.11 PRESERVING INK ANNOTATIONS

Preserving digital ink is an expected requirement of any ink annotation tool, not to mention a necessity for a code review tool due to the annotations being regularly passed back and forward between the reviewer and developer. We have chosen to preserve the ink strokes in 'ink serialized format' (.isf) as it is based on xml (W3C, 2006) and hence allows easier manipulation to include extra xml tags, this is format is explained further in Jarrett & Su, 2002). The support of extra tags is important for us as we need to include extra information in our ink .isf file to store and load the current state of our project variables. Our

system contains information on each annotation within an ink file (e.g. severity and line number of an issue, and all the strokes that correspond to an annotation) this will then need to be loaded along with the strokes locations, in such a way where we can still support the reflow of loaded documents.

5.11.1 LOADING

Whenever any code document is opened a search for an ink file with the same filename is made within the opened documents project directory. If the ink file exists, an ink window is created that contains the code within code window and the ink strokes from the ink file, if an appropriate ink file can not be found, no ink window is created.

When loading digital ink the tool reads all the customized 'issue key' tags of the ink file which gives all the stroke id's, the issues severity and the code line that the annotations belong to. This information allows regrouping the ink strokes to their individual annotation groups, and then reattaches each annotation group to their appropriate code line. The tool also uses the customized tags to populate our annotation array lists in order to support future reflow of the loaded document.

The advantage of having a separate ink file for each code file is that the reviewer only needs to send the ink files of the reviewed material to the developer and not the entire solution (Golovchinsky & Denoue, 2002). This reduces the amount of transfer making it a more efficient process, although this may lead to some consistency problems between the code file and ink file as explained later in this section. Once placed in the correct directory the ink file will automatically load as the code file is opened, it may be helpful in the future to generate a program that automatically places each ink file into their appropriate project directory, but this is not a difficult concept to implement.

5.11.2 SAVING

There are two situations where the ink file can be saved. First, when the ink window is visible and the user hits the save button in the IDE toolbar, both the ink window and its corresponding code window are saved. Second, if a code window is saved and an associated ink window exists, then it too is saved.

The ‘issue key’ tag stores information on each individual annotation within the ink file, this includes all stroke ‘ids’ that correspond to each annotation group, as well as the code line and the severity of the issue. Such that when the ink is loaded the strokes are located in the correct positions and are related to a specific annotation, the issues severity and attached line number also remains consistent to allow the tool to continue accurate reflow.

The naming convention of the ink documents are automatically based on the name of its corresponding code document. For example if a code document is named ‘Class1.cs’ then its corresponding ink document would contain the same name but a different extension type, hence being ‘Class1.isf’. This file is stored in the code files project directory within an ink file specific folder called ‘Ink_Files’, this ensures searching for ink files is more efficient.

5.11.3 MAINTAINING CONSISTENCY

Simply sending the ink files and not the code file is an efficient transfer approach but can also lead to consistency problems. Consistency problems can occur in any situation whenever duplicates of data (code or otherwise) are given to someone else for feedback. The most robust solution is an extreme approach where code is locked such that it can’t be modified while being reviewed. This is an inefficient approach as the developer can not work on separate areas of code while its being reviewed, instead they must sit and wait until the review process is completed.

Consistency problems can occur in this tool as the reviewer generates an ink file then attempts to resolve these issues. The developer’s code file will be different to the reviewer’s code file, and so the generated ink file would relate to incorrect code statements in the developers code version. During a review it is commonly noted that the reviewer should not endeavour to resolve any issues that have been raised due to their lower understanding of the code and its structure, as well as it being an inefficient use of their time (Fagan, 1976). However if they do alter the code to resolve a raised issue then they must ensure to send both the ink file and the code file back to the developer. Sending only the ink file will lead to problems when mapping the reviewers ink file to the developers (now out of date) version of the code file. Another similar problem can also occur, for example if a project contains ink annotations for a specific code file and that code file gets replaced (overridden) but not the corresponding ink file, then again the ink file would be out of date.

Both problems can often be unknowingly caused, the effects of this will cause some ink annotations to be positioned alongside incorrect code statements. A future solution to make this tool more stable against consistency problems involves storing the last edited date of the code file that the ink file was based upon. When the user loads in the ink file a date check can be made between this stored last edited date within the reviewer's ink file and the last modified date of the developer's code file. This will determine if the ink file is based on the same code file that it was generated for. If the two dates are different then the date check fails and a consistency problem exists, then its more than likely some ink annotations may need to be realigned to a new location.

Relocating inconsistent annotations can be handled by storing not only the code line that the annotation is attached to, but also the actual string of code on that line (the code statement). By comparing the stored code statement that the annotation should match-up with and the code statement the ink is currently matched-up with, the tool can determine if that annotation is incorrectly aligned. If an annotation is not matched up with its correct code statement, then the tool could realign the annotation by locating the closest code line that matches the correct code statement. This approach would need further investigation and may be slightly inaccurate, therefore the user may have to confirm each relocation one-by-one.

5.12 SUMMARY

In this chapter we have described our implementation for our IDE code review prototype. We began our discussion of our implementation with an initial overview of this tool, and then went into more detail by discussing in-depth each individual component that made up this tool, as well as the reasoning behind each component. We based our implementation choices on previous research in ink annotation, code review and the user's interaction with our tool. We have obtained encouraging feedback and found some areas of improvement which we included into our final implementation of this prototype. In the next chapter we describe the evaluations that we have conducted on the RCA tool.

Chapter 6

Evaluation

6.1 Introduction

In this chapter we present an evaluation of our RCA tool. The evaluation discusses the strengths and limitations of our design as well as areas of the tool that can be improved and refined before any formal usability tests can be performed. An in-depth evaluation is essential, as this RCA tool has the potential to change the way people perform code reviews in the future.

We begin with an evaluation of the design and implementation using Green and Petre's cognitive dimensions framework (Green & Petre, 1996). This methodology is generally performed by an expert who understands the system well, such as the designer (Blackwell & Green, 2000). We also performed a second evaluation in the form of a post task walkthrough (Dix et al., 2004). This is a widely used observation approach where users evaluate the tool to expose difficulties they faced when interacting with the tool and also where they can propose further functionality ideas. We use both evaluations to find areas where we can refine this tool in order to improve the future usability of this tool. We finish this chapter with a summary of the tools strengths and shortcomings as identified from the two evaluations.

6.2 Cognitive Dimensions Analysis

We have developed a working prototype, however as this project involves a large amount of implementation it is important to find all areas of weakness that need improving before we start adding any task-specific functionality. Of course we want to find areas where our tool does well in relation to this evaluation methodology, although in order to have a tool that is used successfully by many types of reviewers it would be more desirable to highlight and improve the negative areas of the tool.

Cognitive dimensions are a popular method for analyzing attributes, which has been used in many successful visual language tool's to discover a tools strengths and possible areas of improvement (Maplesden, 2001; Modugno et al., 1994; Shum, 1991; Yang et al., 1995). Cognitive dimensions framework is "a broad-brush assessment of almost any kind of cognitive artefact" (Norman, 1991). Generally speaking the cognitive dimension framework involves a set of conditions used for analyzing the usability of programming languages, tools and environments. The terms should be considered as being physical dimensions that are often competing with each other and therefore one benefit may come at a cost. For instance: an increase in functionality may result in an increase of usability difficulty for the tool. In this section we introduce each of the cognitive dimensions individually then relate them against the RCA, and with some dimensions we propose ideas to improve the RCA.

Abstraction Gradient

Abstraction gradient refers to a single concept being generated by a group of smaller concepts. Green & Petre (1996) describes tools as being abstraction-hating (low abstraction), abstraction-tolerant or abstraction-hungry (high abstraction). Therefore, abstract-hating tools involve little initial setup as only a minimal number of concepts are generated, and so are preferred by novice users. However, abstract-hungry tools involve more initial setup as each individual abstraction must be created and combined into the one concept, thus are preferred by expert users as the tool offers more flexibility. As an example, a flow-chart is considered abstraction-hating due to it only involving decision boxes and arrows to other decision boxes, so involves little initial setup. If a tool is used to create a car object such that they must first create dozens of extra objects (wheels, engine, chassis, colour, audio, and vehicle type) before the car object can be constructed, then the tool is considered abstraction hungry.

RCA is essentially a simpler version of a flowchart tool. Rather than having links between two flowchart boxes, this tool contains a linker (line or circle) that joins a portion of code to a group of comments. Therefore the RCA tool has a low abstraction gradient.

However, there is an area that would benefit from a certain degree of abstraction (in this case 'setting up'). A menu that maps error-types to colours could be of benefit to users, meaning review participants could associate a particular issue type to a specific colour. Previously the reviewer (if they wanted) would colour code an issue by selecting it using the selector button from the toolbox, then re-colouring the annotation using the colour dialog menu. Being able to map error types to colours would make it more efficient to classify error types to the exact colour in the future. After issue types have been mapped, the reviewers could simply right-click the annotation, and select an issue type from a shortcut context menu, making it more efficient in the future for users to classify errors with the correct colour.

We can reduce this level of abstraction in the future by having the tool remember the previous mappings when the IDE loads, thus the user won't need to set the mappings every time they use the tool, rather (if needed) they simply edit the existing mappings.

Closeness of Mapping

Developing tools requires mapping between the problem world and a program world (Myer, 1987; Pennington, 1987). The closeness of mapping refers to how closely our tool relates to the real world problem that we are essentially modelling, e.g. how much extra knowledge will need to be learnt before a code reviewer is capable to operate our tool. To do this, each problem must match as closely to the program as possible to ensure similarities between the real world tasks and the tasks performed in this code review tool.

Traditionally, code review is performed by walking-through code, then either noting possible defects (or issues) on a printed hard-copy version of the code or on a separate form. Because we are essentially performing the same task but using digital ink annotation directly over the code, we consider this to be a very closely mapped tool. This means that the reviewers only need a minimal amount of additional knowledge before they can interact with the tool. The users must understand either a circle or line must be used to link a piece of code to a comment. A linker is essential so this becomes an area where the mapping could be improved. For example, we could increase the variety of linkers the tool can support, or even better make the tool intelligent enough not to need them.

Consistency

Consistency in a tool relates to the concepts and operations being carried out in a similar manner throughout the entire task, e.g. each construct is performed in a consistent manner. Green & Petre (1996) use the term 'Guessability', if part of a concept is understood can the rest of the process be easily guessed? The more consistent a tool, the less effort is required by the user to learn the tool, which generally results in the user making fewer mistakes.

Due to the RCA's simplicity we feel this is a consistent tool. For example, once a user can bring up the code's ink window and then realize they must first generate a 'linker' before they comment on a piece of code, they simply continue this process throughout the review process in a consistent manner. We also feel that the ink toolbar is implemented in a consistent way. If the user is viewing code within the code window, clicking any button in the ink toolbar will select that option and make the code's ink window visible for use. The way in which the ink toolbar is used to edit ink strokes (i.e. erase, select, move and re-colour) is implemented with similar functionality to many widely accepted drawing tools e.g. Microsoft Paint (Microsoft Paint, 2006).

Diffuseness/Terseness

Green & Petre (1996) describe a diffuse tool as involving many symbols or ideas to achieve a concept that other tools achieve more compactly, whereas a terse tool is closely mapped to its problem domain and will generally require fewer lexemes to achieve the same result. Classifying a tool as either diffuse or terse is easy, however determining the level of diffuseness or terseness is difficult. To determine the level of diffuseness it's common to compare the tool in terms of similar tools (Maplesden, 2001). The more diffuse a tool is the more functions it may include increasing its usability complexity, whereas a terser tool can be seen as too simple, such that different concepts appear similar to one another. Therefore developing a tool with an equal balance between diffuseness and terseness is preferential.

The RCA tool is closer to being a terse tool, rather than a diffuse one. Generating an issue for a portion of code is as simple as indicating the code portion with a 'linker' and then providing comments alongside this 'linker'. But how terse is this notation? When comparing this tool to existing Microsoft Office ink annotation tools (Microsoft Office Online, 2006) we find their tool simply involves annotating near text with limited editing and colouring coding support. When we annotate text, we understand that each annotation may be unlike previous annotations, some for instance may be much more severe than others and should be indicated appropriately, others may correspond to a different type of issue.

Our tool incorporates severity indicators to differentiate between some annotations as well as colour coding capabilities and in the future the annotations author would also be indicated for use in a collaboration/team scenario. So when compared to the annotation tools seen in Microsoft Office applications, we feel our tool offers more functionality during annotation and should be considered terse overall, but more diffuse than some existing tools. Using severity indicators and colour codes to distinguish between annotations is not strictly needed, and only used to give the user extra annotating options. In the future we would like to give the user extra functionality with their annotations by allowing them to search through each annotation issue, either sequentially or by the annotations severity, we want to also provide the user with some sort of error mapping feature.

Error-Proneness

Errors in general relate to the user performing a task incorrectly, whether they realize there mistake or not. For instance, a logic error somewhere in a program is an unrecognizable error, whereas forgetting a semi-colon at the end of a code line or misspelling a variable name is an accidental mistake (an of error of judgment), we are interested in the latter. When deciding if a tool is error prone, areas need to be determined where the user would make frequent or easy to make errors.

Previously two areas of error-proneness in the RCA existed, the first involved the user thinking they were in inking mode when they were actually in linker mode. In this case the annotation was considered a linker rather than a comment corresponding to a linker. The second area of error proneness was the user not knowing exactly how spatially distant an annotation had to be made, such that their annotation would correspond to a new linker while in inking mode. Both of these areas were realized during our first informal evaluation of our initial prototype. These problems have since been resolved as explained in our implementation chapter.

This tool is generally resistant to accidental errors, though one area of this tool where a user would make an accidental mistake occurs when generating a circle linker that encapsulates several lines of code. If the user generates a circle linker such that they want the vertical midpoint to be attached to a specific line, then the accuracy of this linker is a necessity. This accuracy problem involves the top and bottom extremes of the circle linker where the midpoint of these extremes determines the line number that the issue and its severity should be attached to. If the user generates a circle linker that is accidentally a little offset vertically, then the issue and its severity indicator would be attached to an incorrect line (few lines above or below the desired line). For example, a circle linker was intended to encapsulate only line 13, however as line 14 was accidentally included, the mid-point of this annotation attaches the annotation to line 14, this is shown in figure 6.1.

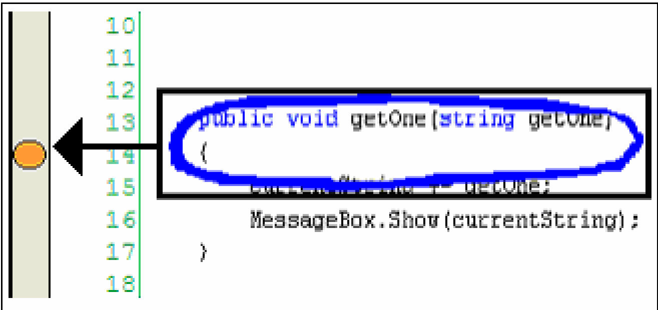


Figure 6.1: Incorrect Attachment of a Circle Linker

In this case the linker would have to be erased and drawn again until the midpoint of the circle is correctly attached to a particular line. Because this may be irritating for some users, it would be convenient for the user to have the ability to drag the severity indicator anywhere between the circles vertical extreme points, such that the issue can be more accurately attached to a line. So from our last example, the user would simply drag the indicator up one line and the annotation would now be correctly attached to line 13.

Hard Mental Operations

Hard mental operations relate to concepts of a tool that require a large amount of mental effort to comprehend. Green & Petre (1996) split hard mental operations into two distinct categories, the first being problematic mental operations that lie at the notational level and generally involves concepts (e.g. conditionals and negatives) that could be expressed in an easier format. The second relates to the difficulty in understanding the overall concept as multiple objects are included, for instance does the concept become harder to understand as more objects are incorporated by the user?

The goal of this tool was to essentially bring a low demanding computerized approach to code review as the old procedure included several manual time-consuming tasks. We feel a lot of the hard mental operations of traditional code review can be eliminated, for instance, consolidating issues into a unique list, and the generation of review reports. Also, as each issue raised is represented in a shared manner with a text-anchor and comment, there is little difficulty when interpreting an annotations meaning. However the area where significant recollection is needed is when the user colour codes an issue to a specific error type. As you increase the possible error-types that an issue can be classified as, it results in both the reviewer and developer having to remember which colour maps to a specific error. As explained in the abstract gradient dimension, this mental operation can be reduced by mapping colours to error types. The reviewer simply classifies an annotation as a particular error type and its colour is automatically set, the reviewer then won't need to know what colour to set the annotation. We must also give the developer some indication of what error type corresponds to a specific colour, so when resolving issues the developer would know that a certain colour would indicate a particular error type. This could be done by having a colour map window being visible to the developer offering fast, easy lookup.

Hidden Dependencies

Green & Petre (1996) explain hidden dependency as being an invisible relationship between two components, whereby one component is secretly dependant on the other component. This relationship can cause major problems as it is difficult to discover this dependency, resulting in users being unaware that a change in one component will yield a change in its dependant component. A spreadsheet's formula cell is a common hidden dependency problem, as its formula cell may be reliant on a combination of values from other 'dependant' cells. This example can get complicated very fast as those dependant cells also take values from other dependant cells and so on, in this case careful consideration is needed when altering the cells formula/value.

The RCA has an obvious dependency between a code window and its corresponding ink window. We had difficulty implementing ink directly over the code window so we split the two windows and made them dependant on one another. We realized this dependency early on in the development stage and wanted to make it easy for the user to identify which window (code or ink) they are working in. We handled this situation similar to how a window's form handles its design view and its dependant code view. We gave our ink window the same name as its related code window, however we append an '[Ink]' string to the end of the ink windows name, such that the dependency is made obvious.

However a more subtle hidden dependency exists with this tool, and relates to the consistency between the reviewer's generated ink files and the developer's version of the code files. Because the code file and ink window are dependant on one another, if a code file is under review and its ink file is generated, then the locations of the ink strokes will be related to the reviewer's code version, not the developer's version. Consequently, during the review, the reviewer can not change their initial version of the developer's code, and the developer can not edit their version of code, otherwise the generated ink strokes will not be based on the same code version. In this case having the reviewers ink file loaded over the developer's code would more often than not result in inconsistent ink locations. The reviewer would have to send the ink file back to the developer along with their new version of the code file. The developer must also ensure their code file remains the same throughout the review, otherwise again the ink strokes will be incorrectly attached. Simply locking the code version until the ink file is generated is a robust solution to this problem, however it's inappropriate, as it is more efficient to allow the developer to continue coding on a separate portion of code in synchronous with the reviewer.

A possible solution to this hidden dependency is to first store the last edited date of the code file within the ink files xml structure. This is so we can check the date of last edit for the developers code file to ensure it is the same as the last edited date found within the code that the ink file was based-on. If these values are the same then there is no consistency problem, however if they differ then it is highly possible some annotations will be attached to incorrect code statements. Each annotation would need to be checked to ensure they appear alongside their correct corresponding code statement, if the annotation doesn't match-up with the code statement then the tool would have to make an intelligent guess as to where the annotation should be relocated. This implies the RCA must also store the code statements text within the ink files xml structure, along with its line number.

Finally as portions of an ink window involve many annotations, it maybe difficult to determine which ink strokes relate to an annotation. This would generally occur if two annotations exist within a few lines of each other and possibly overlap. Highlighting all strokes in the currently selected annotation would solve this problem. If the user is unsure of what ink strokes make up the annotation, they simply tap a stroke to select the annotation, and all strokes within this annotation could appear highlighted.

Premature Commitment

Premature commitment refers to the user being forced to make a decision or complete a task before all necessary information is available, and generally arises when a natural ordering is enforced (Green & Petre, 1996). For example, when linking two components within a flowchart, in most cases the decision boxes must be generated before a link between them can be created. There are cases where the user knows a box will require several links, however these links can not be generated until the boxes are created. When a specific order is enforced it can also be difficult to alter the concept once it's completed. This leads to the user having to delete the existing concept and then repeat each step over, wasting time and distracting them from their overall task.

Unfortunately the RCA suffers from premature commitment. When generating an annotation for a portion of code one must first create a 'linker' before they can add further comments to the annotation. There are two instances where this can cause efficiency problems. First, there may be times where the reviewer has an idea and wants to quickly note down this idea, however before they illustrate their idea a linker will need to be generated. So the reviewer must decide if the comment should relate to a single line or several lines of code, once the linker is determined and drawn, the user may have lost their train of thought. To bypass this problem the reviewer could simply draw any type of linker so they can quickly write down the appropriate comment. Once the comment is written and understood, the user would delete the entire annotation and reconstruct it with the correct linker.
Secondly, the reviewer may want to raise an issue by generating an annotation with a line linker, then after further code reviewing they may decide that the linker should in fact have been a circle. Annotations are built based on the linker stroke which is followed by the comment strokes, because of this the user cannot go back and edit the first element in this annotation due to it being in its second step of creation (adding comment strokes). In this case, the user would again have to delete the entire annotation and regenerate it with a new linker. Both scenarios are inefficient as the annotation must be removed in its entirety, simply to alter one stroke.

The 'linker' is an important part of generating an annotation and we still need it to be the first step in raising an issue. This problem would be reduced by allowing the user to edit the linker stroke, then the reviewer won't have to remove the entire annotation, they would simply modify the type of linker. This would allow the user to create a basic line linker, and once the entire annotation is created they could simply select the linker and redraw it. Modifying this stroke allows the user alter the shape of the linker and also reattach the annotation to a different line. This approach won't eliminate premature commitment, however it does give the reviewer an option to go back and make slight alterations rather than removing and recreating the annotation from scratch.

The optimal solution to this problem is to intelligently attach an annotation to a line without the need of a linker. However, this approach would require further research to determine if users prefer to explicitly select the line to attach the annotation or have the tool 'guess' this line.

Progressive Evaluation

Progressive evaluation is the user's ability to test and evaluate the progress of their partially completed tool at anytime. Progressive evaluation makes it easy for the user to stop at anytime during the use of a tool or notation such that they can check the current state of their work, rather than defer the check until the task is completed.

Because this tool is an extension of Visual Studio's IDE, and simply involves annotating over source code, the only testing performed within this tool occurs when the user debugs and executes the already generated source code. This tool has been designed so that no matter what annotation state the reviewer is in (linking mode or inking mode) the IDE can still debug and execute the underlying code. The ability to debug and execute the code project at anytime is important for a code review tool, such that it may be necessary for the reviewer to generate annotations over a portion of code while they are executing the program. It is possible to annotate code during execution, allowing the reviewer to use break-points to pause the program during execution, then step through each line of code one-by one and annotate simultaneously.

Role-Expressiveness

Role expressiveness relates to how clear a tool describes the role of a particular concept. Expressing an objects role can be improved by using secondary notations, meaningful identities (Teasley, 1994), 'beacons' (Weidenbeck, 1991) or even an explicit description level (Green et al., 1992), such that it is easy to tell what each object in the tool is used for.

The RCA clearly expresses the annotation group that an ink stroke belongs to as well as the role of each stroke within an annotation. A red bounding box is used to express all the strokes that correspond to a selected annotation group. For each annotation group we have a single ink stroke corresponding to the linker, and all other strokes are additional comments. The linker stroke is expressed differently to other additional ink strokes in the annotation group, as the linker stroke has a larger ink width to help the user differentiate them from surrounding strokes.

We feel that this tool also expresses the role of different annotations. First, specific colours are used for our severity indicators to express different levels of severity for an issue. This is done using a coloured Likert scale between green and red (green meaning good/positive, red meaning very bad) as explained in the 'Issue Severity' section of the implementation chapter. Secondly the RCA tool also provides the user an opportunity to colour code their annotations, whereby different colours classify a different error type. Annotations can be categorized differently to other annotations, and each is represented by their severity and colour.

Within our ink toolbar we use informative images to express the role of each button. The user can look at the buttons image and easily understand its function. For example, a pen image implies the user can write over the code, while an eraser image clearly signifies the ability to remove ink strokes. We could also improve the role-expressiveness further by including a description level in the form of 'tool-tip' messages over the toolbar buttons, thus clearly describing the role of each button. A tool-tip message can be included over the severity indicators to express the severity level of that particular colour; and also over the raised annotation to express the errors classification, the severity of the annotation and possibly the annotations author.

Secondary Notation and Escape from Formalism

Secondary notation and escape from formalism relates to the user being able to communicate extra information above and beyond the formal syntax of the tool or notation. In general, secondary notation is beneficial because it allows the notation to be used in such a way that it can improve readability for its users. For example, most coding languages offer implicit forms of secondary notation, including indenting code lines and bracketing code portions to provide an improved layout that is easier to understand. Some coding editors also use specific colouring to express the type of code statement, and by using appropriate variable names and conventions it makes code easier to convey the codes to other readers. Escape from formalism allows a user to supply extra information in a way that is separate from the tools notation, and is generally a more explicit free-form approach that is distinctly separate from the syntax. For instance, program commenting is separate from programming code and allows the user to express exactly what they want in their own native language, without the comments being interpreted as actual source code.

Aside from the secondary notation and escape from formalism that already exist when developing in Visual Studio; we feel the RCA provides a more natural escape from formalism, which is essentially another approach to supply comments directly over code. It allows the user to provide feedback describing the code's functionality and to raise an issue over questionable portions of code, which is distinctly separate from the underlying code statements.

During code annotation the reviewer has a natural approach for providing clear, free-form, informative feedback on issues. If this approach is utilized there should be little need for any further secondary notation for the user's annotations. However if the user provides poor feedback through the ink strokes, or provides a minimal amount of feedback per issue, then the future reader would still have a fair idea of the annotations meaning by understanding the role of the issue in terms of its colour code. Therefore there exists secondary notation support within the RCA tool. For example, if the reviewer simply annotated "fix this problem please" around a five line for-loop, when the developer has to resolve this they may find it difficult to understand the meaning of this comment. But because the reviewer colour coded the annotation issue, the developer would simply realize the annotation is blue and so the annotation can be better understood based on the reviewer's error-type to colour mapping scheme. The developer should then have no problem locating and understanding the exact context of the error.

Currently the RCA has no escape from formalism support in terms of annotation, meaning one can only use this tool to annotate. However extra support for free-form notations could work well while annotating code, especially when the reviewer runs out of annotation room, or they want to supply a more detailed description of the issue.

To support an escape from formalism the RCA could incorporate a pop-up type comment that is attached to a piece of code or an ink annotation. This note could be written in either typed text or digital ink and used when there is little annotation room left. This allows the reviewer to use a limited number of ink strokes over code and they can then provide a more detailed post-it comment to elaborate on the annotation. When this note is displayed it could be expanded to show the comment by hovering the stylus over this note. Rather than over-crowd the ink window with longwinded comments that involve lines of ink, a more compact version of the annotation can be generated and backed up by the post it note.

Viscosity: Resistance to Change

Viscosity refers to the effort involved when making a small local modification to existing work within the tool or notation. Generally people want to see an initial example of an object/concept before being satisfied with the idea, essentially the example 'talks-back' to the user (Black, 1990). This may be an easier approach than designing the complete concept in their heads, and once the concept is designed it should have minimal resistance to refinements. Low viscosity is more desirable and allows the user to create a concept within the tool, and then continue to modify the concept with little difficulty until it resembles the concept they want. A common approach to reduce the viscosity is to include abstractions, such that one change is as easy as changing one of the smaller abstractions, which will then take effect in the overall concept that inherited the abstraction.

When measuring the viscosity of the RCA tool we have to look at the types of changes one could make. Because the tool involves mostly raising and representing issues, the changes that would be commonly made are editing existing annotations. Functions like adjusting an issues severity, appending and deleting ink strokes and moving annotations are easier to manage. Although deciding and selecting the appropriate colour for an annotation may involve slightly more effort. The largest viscosity occurs when a user wants to adjust an annotations 'linker' stroke. As previously mentioned, no simple editing adjustments can be made on this linker stroke. If the linker is not satisfactory then the entire annotation must be deleted and regenerated with a new appropriate linker, this can cause a lot of unnecessary effort. Perhaps in future implementations we could allow the user to recreate a new linker for the annotation such that their existing ink strokes won't require recreating.

Visibility and Juxtaposability

Visibility refers to whether the required information is or can be made readily available. Poor visibility may involve a large number of steps, or even long elaborate searches to simply display an object to the user. Another important component relating to visibility is juxtaposability which is the ability to view two (possibly related) components side-by-side simultaneously. With poor juxtaposability, one would have to rely on their memory or frequently switch between the two items. An example of good juxtaposabilty is being capable of comparing two code scripts for similarity at the same time.

The visibility of the RCA tool could be better. The developer has to look at the ink annotations in a separate window then switch to the code window to edit the necessary portion of code. The developer must then switch back to the ink window to read the next annotation, then again switch back to the code window to resolve the annotation. The visibility problem occurs because the developer has to resolve annotations within the code window and can not actually see the digital ink within the same window. We tried to improve the visibility in the code window by giving the user a cue in the form of a bookmark to represent the lines that correspond to raised issues. As stated previously, we also want to include a search mechanism to increase the visibility between individual annotations. Allowing the user to search for annotations throughout the entire project either sequentially or based on severity, and searching error types would also increase the user's visibility for specific annotations.

However, the visibility problem could be eliminated if the RCA was implemented within a different environment that provided better access to their underlying routines. For instance, the concepts of this RCA tool could easily be integrated into Eclipse's open source environment (The Eclipse Foundation, 2006), and with the extra knowledge and direct manipulation of the underlying routines the code can be both edited and annotated within the same window.

In contrast the RCA has great juxtaposability. Visual Studio allows the user to bring up two windows at the same time via right clicking the code file and selecting 'New vertical/horizontal tab group'. The user can then organize the code window to be on one side of the tab group and the ink window to reside on the other side of the tab group. Perhaps in the future we could have the tool automatically organize these windows for the user. We can exploit this tab group idea further, by amending the code and ink window synchronization. Rather than re-synchronizing the windows each time the user switches between the two, if the windows are in separate tab groups then they would re-synchronize every time the user scrolls the viewing area of ink window. So the keyboard cursor in its corresponding code window could move simultaneously with the ink window, and thus the code views in both windows remain aligned with one another when adjusting the viewing area of the ink window. This works well when both windows are side-by-side, because the user can see the same portion of code in the code window as they can in the ink window.

6.3 Post Task Walkthrough Evaluation

A post task walkthrough is a passive observational evaluation, where the observer simply watches the user perform their specific task, and then the user (evaluator) discusses their interaction and viewpoints with the observer after the task is completed (Dix et al, 2004). This passive approach involves the observer simply taking notes on the evaluator's actions and behaviour as they perform their task to assess the specific tool. The observer is interested in areas where the evaluator seems confused or finds difficulties with the tool and also notes any actions that take longer than expected. These problems are then silently noted and time stamped, then discussed with the evaluator upon completion of their task. This is a different evaluation technique than the previous approach because we must observe the users interaction with the tool in their own environment, rather than the previous approach where the strengths and weaknesses based on the designer's viewpoint.

With this evaluation we observed three types of code reviewers, an industry-based code reviewer, an academic marker and a teacher. First we gave a demonstration on how to use the RCA by performing a code review in front of them involving a basic application containing a few flaws. With this demonstration we made full use of all the concepts and ensured the participant was fully aware of each action we were performing. We also tailored each demonstration to the type of code review the evaluator would perform, simply to explain the RCA based on their environment. For instance, with the industry code reviewer we acted as if we were going to pass this review back to the developer to resolve, for the

marker we based our demonstration on the fact that we were actually marking the application, and for the teacher we presented the material as if we were teaching to students.

Once the user has finished their task, they can describe any possible problems they found when interacting with the RCA. We asked them to comment on the fact that code editing and annotation must be performed in separate windows and we also asked their opinion on the 'linker' concept and the ink toolbar. We are also interested in the users overall experience with the RCA tool, some possible improvements and whether they would find the RCA beneficial to use in the future.

6.3.1 INDUSTRY TECHNICAL REVIEW EVALUATION

The industry evaluator was an experienced test analyzer with two years industry experience and spent the majority of his time testing in the Visual Studio environment. He is now a project manager where the majority of his staff use Visual Studio as their developing environment. The sample application used for the review was a small windows application that resembled a video rental application and contained several errors of different types. The user sample program would supply a members ID, the application would retrieve information about the member from a database and allow them to hire a number of videos from the store. The project manager was asked to perform a code review for this application using the RCA to annotate any issues.

His review team performs code reviews to ensure all logic is correct, that is someone validates each code block (function) such that the correct output is generated for any given input. He performed this task and as an error was found he would indicate the code function and the input, such that the developer can reproduce these errors in the future. The review team would also ensure the project satisfies specific programming standards similar to the standards outlined in (Amoroso et al., 1994; Daily & Foreman, 1984) e.g. proper naming conventions, handled security issues and exception handling. So our evaluator also performed a standards check for this sample review.

It was mentioned that the two separate windows to edit and annotate code would not be a problem for the review team as they don't edit the developer's code when an error is found for two reasons. Firstly, the code files which need to be tested are often large and having the tester find and resolve these errors could take copious amounts of time. The evaluator said "it would be like finding a needle in a hay stack". Secondly, the reviewer did not develop the code and for that reason will not have sufficient knowledge of the underlying code, if they make a small change somewhere, this may have negative effects elsewhere in

the code. The evaluator then backed this up by saying “besides, who will review the reviewer’s changes?” However, in terms of resolving these issues it would be easier if code editing and annotation was performed in the one window, so the developer can see these ink annotations while they resolve them.

A benefit that the project manager brought up was that most development in industry involves projects that are developed as a team, so for most projects, members of his development team will work on the same shared project. Inking over code in this scenario will then give the team members a great way of communicating to one another, which is more distinguishable than current commenting methods. For example, with the ability to merge annotations from each individual ink file into a single shared ink file, the ink could then be used to delegate tasks, for example “Craig, finish off this function please”. Severities could also be used in this instance where the delegated tasks are then given a completion priority.

During a code review, the ability to colour code and attach severities to issues is important. However the current colour coding approach will need to be made more efficient, as currently the annotation will need to be selected using the selector button from the toolbar and then a colour will need to be chosen from the colour menu. It would be more efficient if the annotator could simply right click the annotation and choose from a variety of issue types, this will classify the annotation with a specific colour. Another improvement for this tool would be to allow digital ink over the designer view of a windows form, which will help code reviewers indicate necessary alterations for the user interface.

Overall the user found this to be an exciting and easy tool to use with great potential in an industry setting. Each concept was easily understood and as a whole it offers a great visual representation, which is handy for a team of developers working on different aspects of the same project. However the main drawback for the RCA in an industry setting is due to the ink being attached to specific lines which results in consistency problems. The discrepancies emerge because a code review is performed in parallel with the developers continuous coding. When the ink annotations are loaded into the developers newer version of the code file the annotations will correspond to specific lines within the reviewers code version rather than specific c code statements. When attaching annotations the code statement must also be taken into account. Code statements may have moved to a different code line when the ink file is eventually loaded back into the developer’s code version and as a result the ink needs to be realigned to the correct code statement.

6.3.2 Marking Evaluation

We asked a recently graduated MSc. student from the University of Auckland who has three years experience in marking student's assignments. Choosing a marker with an abundance of experience will give us good feedback and perhaps even several more good improvement ideas. This student has also been shown the Penmarked marking system (Plimmer & Mason, 2006) such that they could imagine the RCA incorporating the same idea in the future.

To perform this marking evaluation we took a first year computer science assignment (written in java) and reproduced it as a C# project in the Visual Studio framework. We then gave this rewritten assignment and its marking schedule to the marker, where he marked the assignment in silence until the task was completed.

The first comment the marker made was rather than giving marks to students, he would rather start by giving them full marks for each requirement then as he found errors he would take some away. I thought this would work well with our marking representation of severities where different severity types deduct different portions of marks. The marker suggested it would be a good idea if deducted marks could be entered into the severities oval, as it will show students the severity of their error and the number of marks deducted.

The second interesting piece of feedback gained from this informal evaluation was that the marker didn't mind that the code couldn't be edited and inked over within the one window. In fact the marker said he would rather only have the ink window appear when he brings up some code, and can forget about the code window altogether. This is due to the marker not dealing with any code editing (assignments being read-only) and therefore shouldn't be changed after the student has submitted their assignment.

The last marking related comment brought to our attention was that the RCA version doesn't have the ability to create 'global comments', these are comments with no relation to any specific piece of code. The marker said this would be particularly good in this situation as there are times where a marker needs to make an overall comment on the assignment that is unrelated to one specific area of code e.g. "Poor naming conventions used throughout", "Poor indentations" or "Well done keep it up". A possible solution for the global comment problem, involves measuring the distance between the annotation and its nearest piece of code. If the linker stroke is over a certain distance, than the linker stroke would be classified as a global ink stroke. Further research would be required to find a suitable approach for reflowing these global comments, they could be attached to the nearest code line and move with the code, or the annotation could be fixed to a specific location on the code.

Overall the marker found the RCA to be a more innovative and intuitive approach when compared to the traditional marking methods. He also mentioned the benefit of placing marks within the severity indicator in such a way that students can see the exact locations where marks were lost. Also the system could use these deducted marks to tally up the marks lost to automatically give an overall score, so the marker won't need to go back through the marking schedule and perform this manually. The students overall mark would be calculated more accurately and would be less susceptible to human error.

The marker also put forward an idea in which the system could recognize a number within a square shaped ink stroke (written anywhere on the code) which can be interpreted as deducted marks. Plimmer & Mason (2006) considered this in the Penmarked tool, however it was decided this approach wouldn't be reliable enough to separate marks over code from annotations. As this was mentioned to the marker, he proposed a solution, to trust the recognition of the ink stroke number it should first be transformed into typed text, such that the marker knows the system has interpreted the deduction correctly. This idea could also be researched further and incorporated along with the ideas within the Penmarked marking tool.

A negative point with this tool is that markers can now make more comments and consequently assignments may take slightly longer to mark. This would however benefit students as their submission results would include more feedback. Another possible drawback with this marking approach is that markers may want to mark assignments at home, which requires them to either have their own tablet PC or have the opportunity to borrow one for marking purposes. There are also times where markers have no choice but to mark assignments on the universities laboratory machines, as some assignments must be tested on machines with a specific operating system and hardware requirements. In both cases a tablet USB pad could provide some benefit, these are similar to mouse pads but with a stylus input device rather than the mouse.

6.3.3 Teaching Evaluation

For the teaching evaluation we asked a senior lecturer from the University of Auckland who has been teaching C# as a new language to second and third year students. This lecturer uses many coded examples that he prepares before class and runs them during class. The lecturer also tends to edit these examples during class to show the effects of similar alternatives. We chose this lecturer due to his widespread use of coding examples during lectures. We asked him to use the RCA tool to give them an 'Introduction to the C# Language' (Manoharan, 2006). We did this to get feedback from the teacher's perspective as a

teaching tool, as well as the feedback from the student's perspective as a learning tool. However, because this tool was developed for code inspection purposes we predicted the lecturer would only make use of the annotation aspect of this tool not the colour coding, nor the use of severity indicators.

The lecturer mentioned he would prepare for a class by annotating over the coded examples prior to teaching the material, and then when code alternatives were thought of during class, the extra annotations can be added. Most of the lecture material in beginners-level programming course contains pages of coding examples. Using the RCA will then reduce the size of the lecture handouts, making them more compact and only containing informative ideas and concepts, rather than countless pages of coding examples. An interesting idea proposed by the lecturer was initially having all annotations invisible, then he can step through each individual annotation one by one, so only one annotation is visible at a time. This can be done with a 'next annotation' button and is similar to viewing the next lecture page, but this time they view the next coded comment in the RCA tool. Separating each annotation in this manner would make it clearer to students which code portion the lecturer is currently dealing with.

The lecturer suggested this tool would be most suitable for first year courses as they contain excessive amounts of coded examples. Whereas more advanced level courses have less coding examples and more detail on concepts. Although there are generally tutorials for these advanced courses that do incorporate many coded examples, and so this tool would be beneficial in these tutoring demonstrations.

As a learning tool, the student said it would make learning code easier where each important piece of code can be easily commented on by the lecturer. Having the comments written directly over the code files rather than copying written material from the blackboard also makes it easier to learn, as they can spend more time listening to the lecturer and less time copying down notes. The student also mentioned he can revisit the coded project in the future and add his own comments, then pass the code around to obtain notes made by his friends.

Overall, both the lecturer and student recognized the benefits the RCA can have in a learning environment. However this would only be worthwhile in a course where the majority of material revolves around code. Both the lecturer and student mentioned a 'prettier' version of the ink strokes could be of benefit, as the readability of a lecturer's speedy handwriting may be difficult to comprehend. Because the lecturer is constrained to only 50 minutes of lecture time and has a lot of material to cover, the handwriting can become quite messy. Digital ink gives us the advantage in the future to use existing beautification algorithms to transform ink strokes into a more readable, cleaner and a more professional representation.

6.4 EVALUATION RESULTS

RCA provides an excellent distinguishable approach for secondary notation over code that is similar to traditional pen and paper annotation. This tool was found to be easy to use, as generating a linker then adding comments is the only concept that needs understanding. RCA also gives the annotator the ability to represent an annotation with added information, by including colour codes and severities. Annotations can also be generated while the reviewer is stepping through code lines during debugging mode. The cognitive dimensions evaluation proved its worth, and the overall feedback from our evaluators during the post task walkthrough was also positive. Our evaluations generated some interesting future improvement ideas to the base layer of this RCA tool, as well as added functionality to serve the needs of scenario specific users.

We present some architectural extensions that can be incorporated into the future version of our base prototype. These are based on efficiency problems from both the cognitive dimension evaluation and during the post task walkthrough. The extra functionality suggested was independent of the review task and therefore gives benefit to all code reviewers. The usability difficulties are then presented to describe common problems our evaluators faced while interacting with our current implementation. Finally we present some scenario-specific ideas that were proposed by our evaluators during their interaction with the RCA tool. These scenario evaluations have helped inform us of the tools strengths and potential improvements within each specific scenario.

6.4.1 ARCHITECTURAL EXTENSIONS

There are some areas of necessary improvements for the RCA tool, noted from both our cognitive dimension evaluation and the post task walkthrough. These improvements must be fixed before any formal evaluation can be performed and to prevent users from reverting back to tradition review methods.

The major architectural problem from our cognitive dimension evaluation was the visibility problem, where the RCA doesn't provide inking directly over the code window. We noted earlier that this was our most favourable approach, but it was difficult to implement due to the lack of scrolling event indications, which is one of the most important events we needed to handle. When the code window scrolls up/down, we have to intercept this event in order to also scroll the digital ink, so the annotations remain correctly attached to the appropriate portion of code.

The majority of our evaluators didn't appear to be bothered by coding and editing in separate windows, however this inconvenience would mostly affect the developers as they read ink then resolve the annotations. This problem can be reduced via grouping the windows on separate sides of a tab bar, and then have the two windows move in synchronous with one another.

There are three minor areas of improvements that were noted from our cognitive dimension evaluation. First, a mapping window is needed that allows annotators to create an error-type and associate a specific colour to this error. This will eliminate the need to remember what ink colour relates to a certain error-type, and benefits both the reviewer when classifying error-types and the developer when understanding each specific colour. Second, there is no support for linker editing. This problem occurs when a linker's midpoint isn't accurately attached to the appropriate code line, or the shape of the linker needs changing in the future. The user must then delete the entire annotation and then recreate a new linker and rewrite the original comments. Simply keeping the original comment strokes and being able to replace the linker stroke with a new linker would make linker editing more efficient. Third, there are several areas of hidden dependencies. The first hidden dependency involves the consistency between the code file and its corresponding ink file, where there exists the possibility that the code lines in the ink file are different to code lines in the code file. This occurs as the code is edited while at the same time the original file is being inked over by the reviewer. This can also occur if the reviewer edits their version of the code file while inking, and then doesn't notify the developer of these changes. When the ink file is loaded into the new version of the code file there is the possibility that the lines in the code file are different than the lines that the ink annotations were based on. Rather than attach annotations to a specific line, they must be attached to a line based on both the codes line number and the codes text. The second hidden dependency occurs when there are several overlapping annotations existing around nearby portions of code. This problem results in some difficulties when determining the ink strokes that correspond to a certain annotation. A solution to this problem was making all strokes of the currently selected annotation appear in a highlighted version of their current colour, this will ensure the current annotation strokes standout from other ink strokes.

Our post task evaluation presented four scenario independent improvements for the RCA tool. First, common knowledge suggests, the more someone uses a computerized tool, the more they rely on short-cuts (Dykstra-Erickson & Curbow, 1997), it was mentioned the RCA tool should also incorporate short-cuts. Our evaluators agreed on the need for a short-cut menu in terms of a drop down list when right clicking (of the stylus) over the ink window. This will help users perform their tasks more efficiently. This drop down menu could allow users to proficiently switch from the ink window to the code window using the 'view code' option.

The menu could also make it easier for reviewers to classify annotations into a specific error-type as they would simply select an error-type from a list shown in the drop down menu, rather than select the annotation and then the colour using the colour dialog menu. It may also be user friendly to include an 'undo' option in this menu to allow the user to cancel their previous action.

Second, one user during our post task evaluation preferred using margin bar linkers as apposed to circles, so perhaps the tool could incorporate more styles of linker strokes. Increasing the variety of linkers would improve the tool by making it similar to the annotation freedom seen within traditional pen and paper annotation. The evaluators also thought global linkers would be of some benefit, such that the linker is independent of any specific lines of code, this would be great for markers to express general comments. Global comments would also add purpose during industry reviews to express common mistakes and overall feedback, as well as for communicating to team members. Further research into attaching global annotations and the ability to intelligently attach annotations to code without the need for linkers may also be useful.

Third, the users thought their linker strokes looked messy and unprofessional. To improve this we could run an automatic stroke smoothing filter on each linker to remove any jitters from the stylus or mouse (Duda & Hart, 1973), or consider beautifying these strokes (Plimmer & Grundy, 2005) to give a more formal and respectable appearance for future readers. It was also proposed that we investigate smoothing not only the linkers but also the comment strokes. As some tasks involve time constraints (in particular lecturing and marking) and can affect the cleanliness of the annotators handwriting, this is especially notable with lecturers as their notes are usually seen to be messy and sometimes even illegible.

Finally, it was mentioned by both our industry participant and our marker, that an automatic summary of issues in a report would be a handy feature. At the end of an industry based group meeting, a report summarizing the presented issues and the agreed upon severities could be automatically generated. This would save the meeting moderator time, as this is previously manually performed by the moderator. After the completion of a marked assignment, the automatic generation of a report illustrating all the markers annotations and the number of marks that were deducted would also be of some benefit to the student. Rather than the student looking through all the loaded ink files to find where they went wrong, they can simply refer to the summary sheet for a simplistic view of their mistakes.

6.4.2 Usability Difficulites

From the post task walkthrough the major difficulty all evaluators encountered involved our ink toolbar. First, it seemed that each user was uncertain whether the button on the ink toolbar was selected. This was noticeable during the evaluation as the user clicked the button several times to ensure the button was pressed. This toolbar needs improving such that the user can easily observe at any time if a button within this toolbar is selected or not. A tap on a toolbar button should result in the button remaining 'pushed-in', highlighted or even viewed in black and white such that it's clearer to the user that the button was pressed.

Second, the button images in the toolbar are clearly represented. However contrary to our belief the users were unsure of what each button performs, perhaps a tool-tip caption would be useful, that describes the buttons functionality when the stylus is over the button. Also the buttons backgrounds are not transparent and so the images don't blend into Visual Studio well, resulting in them looking distinctly separate from the underlying IDE. To handle both problems, we might look at using a toolbar extractor program (Software Catalogue, 2006) to extract all toolbar images within Microsoft Office's Ink Toolbar (Microsoft Office Online, 2006). This will allow our ink toolbar to resemble the same ink images commonly used in Microsoft's ink annotation toolbar, which is becoming a more universal, and hence more intuitive for our users to understand and recognize.

6.4.3 Scenario Specific Extensions

From our post task walkthrough of our industry-based evaluator, the most interesting task-specific extension mentioned was the need for merging ink strokes onto a shared code project. From this evaluation we agreed on two situations where merging ink from different versions of the same file would be of benefit. First, as some development is performed in teams, where individuals would work on a separate portion of the same code project or even the same file. Then annotations over one version should be displayed over all versions, for all team members to see. Second, as the developer submits a version of code to the reviewer for feedback, it is efficient for the developer to continue extending their version while their code is being reviewed. Once the review is complete, all annotations should be merged in with the developer's latest version of code, so the developer can still work on their version while the old version is being reviewed.

Our marking evaluator mentioned along with the functionality of Penmarked (Plimmer & Mason, 2006), it would also be good to have numbers in the severity indicators that can be interpreted as deducted marks. Another marking related extension the RCA tool could support is mark recognition, this means the tool would recognize a number within a square box as being a mark deduction, and this would be translated into typed text and deducted from the student's mark. The reason for first representing it as typed text is so the marker knows the tool interpreted the value correctly. Once the marking is complete, rather than the user having to go through and calculate the student's final mark, the tool could scan through all the mark deductions and automatically generate an overall score.

Our last code reviewer was a lecturer who brought up an interesting feature, the ability to show each annotation individually. Having this feature the lecturer could prepare ink issues before class that correspond to portions of code, during the class he can describe annotations one-by-one, making it clearer to students what portion of code is being discussed. The lecturer can then easily step through each code annotation individually much like discussing one code extract per lecture handout page.

6.5 Summary

This chapter described two evaluations, both evaluations helped identify a number of strengths and weaknesses of this RCA tool. The first evaluation was based on a developer's perspective in terms of applying the cognitive dimensions framework, we gave our feedback on how our tool rates against these dimensions, and we also recommended potential usability improvements. The second evaluation was a post task walkthrough where skilled code reviewers gave both task-specific feedback as well as general improvement ideas which we could also incorporate in the future. In the next chapter we give an overall discussion of the final implementation for this prototype, we also discuss our experiences and issues with this tool.

Chapter 7

Discussion

7.1 Introduction

This chapter first identifies the overall problem that this thesis solves and the adopted approach to solving it. The background for the RCA tool and a variety of code review scenarios that influenced our final design are then discussed. Chapter 7 concludes by discussing first, the implementation of the components that combine to give our first code annotation prototype, and second, and an outline of the improvements to be made to the base prototype that were obtained from our evaluation. Finally, we elaborate on the strengths and weaknesses of the RCA tool.

7.2 Tool Discussion

This thesis investigated the requirements for integrating digital ink annotation within an IDE for the purpose of code review. Digital ink annotation has been successful for documenting issues within digital text documents (Janssen, 2005; Microsoft Office Online, 2006; Schilit et al., 2003). It is of interest whether this success can be seen when annotating code documents. A computerized approach to code annotation will eliminate the manual tasks that are performed during code review and also gives the reviewer a natural annotation experience similar to tradition pen and paper. Ink annotation is useful within an IDE as an effective code review approach, as feedback can be written directly over code in a distinguishable format. We explore this further by designing, developing then integrating an ink annotation tool seamlessly within Visual Studio. The result is code can be annotated by a reviewer while retaining all the benefits that the IDE provides. The annotation tool was completed, and then underwent two separate evaluations to identify necessary improvements and ensure reviewers won’t revert back to their previous review approach.

An extensive literature review was presented to support this work. Firstly, a study into traditional pen and paper research indicated some interesting statistics on the majority of annotations used (Bargeron & Moscovich, 2003; O'Hara & Sellen, 1997) and the functions to why these annotations are made (Marshall, 1997). Because we are constructing a tool to benefit code reviewers it's important that once the code is reviewed the feedback must be easily understood by the developer, so we investigated how to generate annotations in a shared environment (Marshall & Brush, 2004). Using the common types of annotations and the characteristics that make an annotation easily interpretable, we designed a linking concept to entice annotators to link a portion of code with a comment.

Digital ink annotation concepts were reviewed such that the advantages seen with digital ink could be applied into this annotation tool. The first advantage and the most difficult requirement to implement was to provide the capability of editing the underlying code and have all ink strokes reflow to reflect these changes (Bargeron & Moscovich, 2003, Brush et al., 2001; Chiu & Wilcox, 1998), this was supported by implementing vertical ink reflow. The second advantage was the ability to provide clean and easy editing of ink strokes, an ink toolbar was created to provide this editing capability in terms of deleting, selecting, and re-colouring annotations.

Because this project investigates the effectiveness of raising coding issues directly over an IDE, we reviewed existing code inspection procedures (Fagan, 1976; Johnson, 1994) and code walkthroughs (Wiegers, 2001) noting their similarities and differences. This provided an understanding of how code reviews are performed and how coding issues are recognized and represented. Issue line numbers and severities were considered necessary when creating an issue. Several time consuming tasks were also brought to attention, these can be eliminated with a computerized approach, and include consolidating issues from multiple reviewers, and representing the issues in an inspection report.

Several tools that support feedback over different types of digital documents were reviewed and they provided some interesting ideas and concepts. XLibris (Schilit et al., 1998) supported annotation of fixed documents while incorporating the tangibility features that are seen in traditional pen and paper annotation. Golovchinsky & Denoue (2002) extended XLibirs where existing ink strokes now reflowed as the underlying layout and font of the document was resized. An ink annotation tool was developed to allow ink annotation directly over web pages, by embedding the ink characteristics into the web documents html elements (Ramachandran & Kashi, 2003). Feedback in terms of symbols and labels has shown to be an effective approach for marking digital versions of student's assignments (Heinrich & Lawn, 2004), and Penmarked (Plimmer & Mason, 2006) provided similar feedback over student's assignments but with digital ink annotations.

As this tool is developed 'seamlessly' within an IDE, we looked at several tools that have also been integrated into an IDE. Penmarked (Plimmer & Mason, 2006) allowed annotation over code, this was not integrated into an IDE, although the benefits of integrating this platform into an IDE are easily visible. We described an add-in for the Eclipse framework that efficiently locates all dependencies for a code object (Robillard & Murphy, 2002). The Freeform tool (Plimmer & Apperley, 2003) was integrated into the Visual Basic 6 IDE and allowed ink annotations to be transformed into a user's design form, and 'Blog Reader' (Conwell, 2004) was integrated into Visual Studio 2005 as a way to check web blogs while developing code within the same application.

A description of several code review scenarios was then presented discussing the current ways teachers, markers, industry reviewers, peer reviewers and developers perform their specific task. We then propose how an ink annotation tool within an IDE can be used to perform the same task more effectively and we constructed a list of features that each reviewer would benefit from. Due to the time constraints of this project these features were filtered into a core set of requirements, these include extending the IDE, digital ink support, reflowing ink, modifying ink, indicating an issues severity and preserving ink.

7.3 IMPLEMENTATION DISCUSSION

The RCA tool was implemented by extending the Visual Studio 2005 IDE and it runs once the environment starts up. Free-form ink annotation over code was provided using a supplementary ink window for each code window, as previously mentioned the IDE does not expose scrolling event handlers. However, with an open source IDE it should be possible to simply attach our ink window directly over the code window and have the ability to edit code and ink within the same window. This would be done by intercepting scrolling events to have the ink strokes move alongside the code, and this would be our optimal design choice.

Each annotation is attached to a specific line in the code file and this is based on the position of an annotations first stroke, which is called the 'linker'. The tool then switches from 'linking' mode to 'inking' mode and is signalled to the user by a change in mouse pointer and by the inclusion of a red bounding box encapsulating the current annotation. A linker can be interpreted as either a line or a circle, and once created further comments can be included with this stroke as part of the issues annotation group.

A line linker is attached to the line closest to its start point of the stroke, and a circle linker is attached to a line closest to its vertical midpoint. To create a new annotation, the user can wait three seconds for the mode to switch back to 'linker' mode or start a new stroke outside the annotations bounding box. To amend strokes to an existing annotation, it must first be activated by clicking on one of its strokes when in 'linker' mode, and then new strokes will be added to this annotation group.

As code is added or deleted in the code window, the RCA will refresh the code in the ink window and reflow the locations of existing ink annotations such that they correctly realign with the underlying code context. Only vertical reflow was considered important when supporting ink over code documents, as code is generally one statement per line and also because a slight horizontal offset of ink doesn't bother most users (Bargeron & Moscovich, 2003). Ink reflow was implemented by first consolidating strokes into a logical group based on spatial and temporal properties, which is then attached to a specific line based on the annotations linker stroke. Finally all 'TextChanged' events are intercepted and the ink annotations would move accordingly to whether the text was added or deleted. The tool also includes a variety of ink editing capabilities on these annotations, including selecting, resizing, copying, moving, deleting and re-colouring the annotations.

We wanted to allow the user to personalize each annotation. The RCA gives the user an opportunity to supply extra characteristics for each annotation, such that annotations can be interpreted differently to others. Firstly a colour-coding scheme was proposed, where a specific type of coding issue could be represented by re-colouring the annotation in its own unique colour. Secondly, a coloured indicator is associated with each annotation (green, yellow, orange and red) and resides in the margin where it is used by the annotator to indicate the seriousness of an issue. This indicator then gives the developer an idea of the annotations priority. Future versions of this tool would supply a user-friendly searching function, where users can quickly view the next annotation within the project either by severity or sequentially. Once the reviewer has indicated the type of error, a search based on error types could also be performed. Preserving digital ink was handled within Visual Studio, when the code or ink window is loaded or saved so too is its corresponding window. When dealing with annotations made in a team environment, it may also be desirable in the future to indicate the annotation's author, so that any issue quarrels can be directed at the one who raised the code issue.

It is also necessary to sustain the same viewing area across both the code window and with its corresponding ink window. As a user reads an annotation in the ink window then switches to the code window to resolve the annotation, the code window will automatically adjust its viewable code portion. This is so the user will see the same lines of code in the code window as they saw in the ink window. Therefore the code window will not need to be scrolled to find the appropriate annotation shown in the ink window.

Several code review concepts were also investigated and would provide some benefit in future versions of this project, but overall we feel the current implementation of this tool caters for the basic needs of all reviewing scenarios independent of their task. However, before we can extend this implementation to cater for a specific type of code reviewer it would be an advantage to first implement all general tool improvements and resolve all usability difficulties discussed in our previous chapter. Only then can this tool be extended with the scenario-specific features discussed in the scenario chapter and from the feedback obtained from our post task walkthrough.

7.4 EVALUATION DISCUSSION

Chapter 6 presented two informal evaluations. The first is from the developer's perspective in the form of analyzing the cognitive dimensions of the RCA. This highlighted several areas where the implementation was successful, and other areas where enhancements were needed to which we proposed possible solutions. The second evaluation was a post task walkthrough performed on three independent review participants. They were a project manager from a successful industry development team, an experienced marker, and a university lecturer. This usability evaluation like the first evaluation above gave positive tool feedback, difficulties and improvement suggestions.

From our initial experiences and the remarks from our evaluators during the post task walkthrough we are happy with the tool and feel it works effectively within our chosen IDE. The individual concepts were easy to understand and creating and correcting issues were equally straightforward. More specifically, industry-based code reviewers would use this tool to raise issues affecting improper use of coding standards, indicating logic errors and ensuring client specifications are met. This tool also poses as an effective means to promote communication among developers working on the same development project.

This tool would benefit students as their assignment marks would contain more comments explaining areas where marks were deducted. The future version of this tool could incorporate ideas from the Penmarked tool (Plimmer & Mason, 2006) as well as recognize deducted marks. This tool could then tally up marks and email them to students automatically, making the marking process more efficient. Finally, it was realized that this tool would be most suitable as both a teaching and as a learning tool for beginner programming courses, and tutorials that rely on coded examples.

The main usability difficulty found by our users involved the ink toolbar, where users were unaware if the clicked toolbar button was correctly activated. To better represent a selected button, it would be necessary to have the clicked button appear in black and white, or even have the button retain its pushed-in effect, perhaps also an audio sound.

There are several improvements to the base functionality of the tool that must be resolved before we can add scenario-specific additions. One user mentioned the toolbar buttons were not consistent to the existing buttons in the IDE as they didn't have a transparent background. Incorporating a right click drop-down menu could include 'view code window' that will make switching between ink and code windows more efficient. The drop-down menu could also be used to classify annotations to a specific error type, and an 'undo' option could also be included. With time constraints on lectures it is important to annotate explanations quickly, this rushed action may result in comments which are unclear. Beautifying ink comments may therefore be a useful feature when teaching, as well as in other scenarios where users desire their ink strokes to be represented in a clean professional manner e.g. industry code reviews and when marking assignments. Finally, there are two improvements that would allow the annotator to be less precise when constructing a linker. First, allowing the user to edit the linker strokes without deleting its comments. Second, allowing users to move the issues severity indicator between the top and bottom extremes of the circle linker when the indicator appears a little offset will eliminate the need to recreate the linker stroke in order to adjust its attached line.

Once the noted improvements are resolved from our two evaluations, then formal evaluations of this tool can be carried out. These evaluations would determine the effectiveness of this tool for both annotating coding issues and as a learning tool, and can then be compared to traditional approaches. It would also be interesting to investigate the efficiency of this tool when performing a review task compared to the reviewers previous approach.

7.5 SUMMARY

There are several standout positives for this method of code review. First, the comments will visually stand out from the underlying text because of the natural and distinguishable representation. Second, because we are using computerized annotations they are easily modifiable and we can also store additional information alongside each issue, for instance the RCA currently supports severity indicators and colour coding. Third, code annotation is performed directly within the coding IDE, so the tool retains all existing functionality available with the IDE.

However, we feel the main weakness of this tool was being unable to create a transparent window directly over the code window such that code can be edited and annotations can be seen within the same window. Yet it seemed from the evaluations that this problem was not as severe as originally thought. In fact most reviewers (markers, industry reviewers) only deal with read-only copies of the code, and it's only when resolving issues where this would be seen as a hindrance. This could also be a problem if the developer decides to comment with digital ink during development, as he/she would want the comments to be visible when making changes to existing code. We did however discuss how this problem can be minimized, we synchronized the viewing area of each window and in the future we could automatically organize both the code window and the ink window side-by-side so they are both visible at the same time.

In this chapter we discussed our tool, its implementation and the research evaluation along with the strengths and weaknesses of the RCA tool. The next chapter concludes this thesis by presenting the contributions of our work and by proposing future work to satisfy individual review scenarios.

Chapter 8

Conclusion

8.1 INTRODUCTION

This chapter presents a summary of the main contributions of this thesis and then provides several areas for potential future work. Finally a general summary is presented to conclude this thesis.

8.2 CONTRIBUTIONS TO THIS THESIS

The RCA integrates digital ink annotation into an existing code environment for the purposes of reviewing code. Therefore this thesis contributes to three main areas of research, in particular supporting digital ink annotation over source code, a computerized tool for code review and integrating tools into existing IDE's. These contributions are partitioned and individually described below:

Sharable annotations that can be understood by other readers. The major component of the RCA and the main contributing factor to this thesis is the ability to support digital annotations directly over code within an IDE. These shared annotations combine both a point of reference in close proximity to code and an annotator's attached comment.

Computerized support for code review. The RCA tool provides the ability to perform code review within the IDE by enhancing the code environment with ink annotation. This extra support offers a computerized code annotation approach that can reduce efficiency problems seen from performing traditional code review.

An effective approach to highlight issues during code review. This approach contributes to code review by providing several advantages over the traditional review approach. It allows the annotator to directly reference the portion of code with distinguishable ink, and provides additional features where annotations can be prioritized and colour-coded. Annotating code can also be performed while executing the reviewed program, without switching between separate applications.

Grouping annotations for reflow support in code documents. While our overall reflow methodology wasn't our own idea, we did take a slightly different approach when combining ink strokes into logical annotation groups. The RCA provides the user with a visual representation of how each annotation is grouped. We feel this concept gives a clearer indication of the temporal and spatial constraints of grouping ink, such that annotators become aware of how and why their ink strokes were grouped. As a result this concept removes any accidental errors or ambiguities that may be seen when grouping ink strokes in other annotation tools.

Integration capabilities of Visual Studio. We feel the RCA is extremely beneficial to all developers who use Visual Studio, as it demonstrates Visual Studio's ability to support customized functionality of a complex tool. Tools can be integrated directly within this IDE, eliminating the need for switching between several applications while developing. This is not specifically related to Visual Studio, other IDEs that have integration capabilities may also be used to support additional functionality.

8.3 FUTURE WORK

The short-term future work includes general improvements in order to have a complete base prototype, and these improvements were presented in our evaluation chapter. The future work in this chapter relates to further research areas that can be explored, in terms of supporting specific code review scenarios. Further investigations can also be used to improve the reflowing of digital ink over code. Finally the RCA should undergo a formal evaluation in the future.

The possible future enhancements of the RCA tool include investigating and implementing the requirements for several scenario specific extensions. Such that the RCA tool can provide specialized functionality depending on the type of code review task performed. These scenario specific extensions include the following:

Support for industry-based code review. An entire review procedure similar to the formal review process described by Johnson (1994) can be incorporated into the RCA tool. The tool can allow individual reviewers to conduct private reviews, then during the public review all private issues can be collected and consolidated into unique annotations and can be made available for all to comment on. Timing support can also be incorporated to accurately calculate the duration of each review, where each minute will only be counted if the tool experienced user interaction. During the group meeting, raised annotations can be discussed and perhaps more annotations can be made. After this meeting an inspection report can be automatically generated containing a well-presented list of all annotations that require resolving. This report would be emailed back to the developer along with the digital ink files for issue resolution.

Support for marking specific functionality. Assignment marking ideas found in Heinrich & Lawn (2004), Myers et al. (2004) and Plimmer & Mason (2006) can be incorporated, so that marking and annotating can be done directly within an IDE. Several methodologies could also be explored to support mark recognition directly over code, much like the idea explained in our evaluation chapter.

Support for teaching specific functionality. Support for expressing more clearly each individual annotation will indicate to students which portion of code the lecturer is currently dealing with. Further requirements that benefit a teaching environment could also be investigated.

Support for collaboration specific functionality. Several universities provide tablet PC's to students during class-time, and provide them with the capability to collaboratively share lecture notes during class-time (Huang et al., 2003; Moran et al., 1997; Simon et al., 2004). This collaboration support could be included into this tool, such that students can share class notes.

In terms of the future work that can be done with ink annotation over digital documents, there are still many areas that can be investigated. The major challenge that this project has overcome was supporting reflow of digital ink annotations, it paves the way for further research in this area. The majority of ideas that have emerged from this implementation revolve around attaching annotations to code as well as features relating to the representation of annotations in both a textual and coding review environment, these include the following:

Linkers. During our usability evaluation it was the markers desire to use a vertical margin bar to specify a portion of code statements rather than a circle linker. Perhaps this could be investigated further, by allowing the user to make free-form linker annotations, so that an experiment can be made into observing the majority of annotations types used by code reviewer to reference code. A system could be investigated

that recognizes these common annotation types which could then be provided as linker strokes within the RCA tool, this would give the annotator a variety of different ways to specify a portion of code.

Resizing Annotations. Golovchinsky & Denoue (2002) stretched and compressed its digital annotations as the digital documents font and aspect ratio increased and decreased. Perhaps the ink annotations within the RCA tool could also adjust its size to match the underlying proportion changes. The RCA deals with vertical reflow and as a result the tool could stretch the height of circle linker strokes and other future linker types that encapsulate a portion of code. These linkers would need to include the upper and lower vertical code line boundaries, so as code is added between these extreme points the linker stroke would adjust vertically to continue to include these top and bottom code statements.

Explicit or implicit linkers. An alternative approach to attaching ink to portions of code could be investigated where annotations can be generated without explicitly linking a comment to a portion of code. The RCA could intelligently attach comments to a portion of code lines or an individual code line. This will mean annotators can still use linkers (circles, lines, brackets) to specify a portion of code but it is not required. It may even be the case that annotators or annotation readers prefer linkers, as they explicitly specify a portion of code, rather than have the tool implicitly guess the code line to which an annotation corresponds to. The user’s opinion on each approach could also be evaluated.

Global Comments. Another possible research area in terms of this linker concept is providing annotators with the ability to make global annotations. Global comments are those that are not related to any specific piece of code and so they do not require a linker stroke. These global annotations can be used to give positive and negative feedback back to the developer. An investigation can be conducted to determine whether users want their global annotations to reflow with the document, or have them fixed to its original location.

Beautifying ink strokes. The lecturer mentioned the possibility of beautifying ink strokes. Beautifying ink strokes is helpful due to the fact that a lecture is restricted by time, meaning lecturer’s handwriting is often untidy. Perhaps a study can be performed on the effectiveness of beautifying code annotations or even investigating the user’s reactions to beautified ink.

Maintaining Ink Consistency. Investigating an accurate approach to handle consistency problems between the ink files and their underlying document (code or otherwise) is another interesting feature that any ink annotation tool could benefit from. A check would first need to be performed that determines whether the ink file was generated from the same underlying document that the ink is being loaded into. If this check results in the ink file being loaded into a different document version that the ink was based on, then there is a high chance some annotations may not be correctly aligned to the appropriate code statement. In this case some annotations may need to be relocated to a nearby code statement that the annotation was originally attached to.

Handling duplicate annotations. The last area of possible future advances involves a tool that realizes and manages duplicate annotations obtained from multiple reviews of the same code file. A duplicate is any annotation that is attached to the same line but in different ink files. For an annotation to be considered a duplicate with another annotation, both annotations must be attached to the same code line and perhaps even involve the same linker type and error classification. If a duplicate is found, the tool could just use the first annotation and remove all others. Another possible approach is to stack all duplicate annotations into a logical group and have the top-most annotation viewable. When the next annotation in the stack is desired then the top annotation could be sent to the bottom of the stack, such that the next annotation is displayed. These concepts could also be applied when merging ink annotations from multiple versions into the one shared file.

Finally it is clear that a formal investigation is needed that evaluates the effectiveness of digital ink as a method to describe code. Our usability tests gave us a good insight into the future of this tool and the positive feedback showed promise for continued extensions of this tool. However, we plan to implement the future scenario specific extensions, and then allow markers, teachers and industry-based project teams to use this tool for a period of time. This type of evaluation would provide some comprehensive feedback for further enhancements. A formal evaluation that compares the efficiency of this tool compared to the reviewer's previous approach would also be valuable in finding areas for improvement.

8.4 FINAL CONCLUSIONS

In this thesis we have investigated the user and technical requirements needed for ink annotation within an IDE to support code review. Ink annotation is an effective way to record comments on documents, either on paper or in a digital environment. Program code differs from text documents in that it is non-linear, this suggests code annotation will be more effective when performed directly within the IDE as the IDE supports appropriate code navigation.

We began with a comprehensive overview of both ink annotation and code review. Then we presented some related tools that support annotation over a variety of documents, followed by some plug-ins tools that give coding environments extra functionality. This literature review, combined with our investigations of several code review scenarios, gave us the chance to formulate a set of key requirements to provide effective tool support for code review practices. Next was the design step, where the list of requirements was listed and prioritized into a smaller set of core requirements, these were discussed in more detail and then the ideas were used to implement a prototype plug-in called RCA. RCA is the first tool that provides ink annotation feedback directly over code within an IDE. Finally two evaluations were performed on this RCA tool, and the results have suggested this approach with appropriate refinements has great potential to support code annotation and shows promise in both academic and commercial environments.

Code review was a highly utilized procedure when it was first described by M. E. Fagan. Back then code was sequential and could be read like a book. However, as code became more object-oriented, it became increasingly difficult to interpret and annotate. Now with the use of the RCA tool, code review may once again be established as the most significant procedure within software engineering.

BIBLIOGRAPHY

Adler, M. J., & Van Doren, C. (1972). *How to Read a Book*. New York, NY: Simon and Schuster.

Adobe (2003). *Adobe Acrobat 6.0 Professional*. Retrieved 17/12/03, 2003, from http://www.adobe.com/products/acrobat/main.html

Amoroso, E., Taylor, C., Watson, J., & Weiss, J. (1994). A Process-oriented Methodology for Assessing and Improving Software Trustworthiness, *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, Fairfax, Virginia, pp. 39-50.

Anderson, R., Anderson, R., Simon, B., Wolfman, S. A., VanDeGrift, T., & Yasuhara, K. (2004). Experiences with a Tablet PC Based Lecture Presentation System in Computer Science Courses, *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, pp.56-60.

Arthur, L. J. (1993). *Improving Software Quality: An Insider's Guide to TQM*. New York: John Wiley & Sons.

Atkinson, R. K., Derry, S. J., Renkl, A., & Wortham, D. (2000). Learning from Examples: Instructional Principles from the Worked Examples Research. *Review of Educational Research*, 70(2), pp. 181-214.

Bargeron, D., & Moscovich, T. (2003). Reflowing Digital Ink Annotations, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Ft. Lauderdale, Florida, pp. 385-393.

Basili, V. R., Selby, R. W., & Hutchins, D. H. (1986). Experimentation in Software Engineering. *IEEE Transactions on Software Engineering*, 12(7), pp. 733-743.

Black, A. (1990). Visible Planning on Paper and on Screen: The Impact of Working Medium on Decision-Making by Novice Graph Designers, *Behaviour and Information Technology*, 9(4), pp. 283-296.

Blackwell, A. F., & Green, T. R. G. (2000). A Cognitive Dimensions Questionnaire Optimised for Users, *Proceedings of the 12th Psychology of Programming Interest Group*, Cozenza, pp. 137-154.

Brush, A. J. B, Bargeron, D., Gupta, A., & Cadiz, J. J. (2001). Robust Annotation Positioning in Digital Documents, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Seattle, Washington pp. 285-292.

Chen, C. Y., (2005). *Lecture 3*. Retrieved 30/11/05, from http://www.cs.auckland.ac.nz/compsci230s2t/lectures/Yen/cs230-lecture03.pdf

Chillarege, R. (1999). *Software Testing Best Practices*. Retrieved 25/7/06, from http://www.chillarege.com/authwork/TestingBestPractice.pdf

Chiu, P., & Wilcox, L. (1998). A Dynamic Grouping Technique for Ink and Audio Notes, *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology*, San Francisco, California, pp. 195-202.

Ciolkowski, M., Laitenberger, O., Rombach, D., Shull, F., & Perry, D. (2002). Software Inspections, Reviews & Walkthroughs. *International Conference on Software Engineering*, pp. 641-642.

Colen, L. (2001). *Code Reviews*. *Linux Journal*, 81(11), Retrieved 25/7/06, from http://portal.acm.org/citation.cfm?id=364693#

Collofello, J. S. (1987). Teaching Technical Reviews in a One-Semester Software Engineering Course, *Proceedings of the 18th SIGCSE Technical Symposium on Computer Science Education*, St. Louis, Missouri, pp. 222-227.

Conwell, J., (2004). *BlogReader: Add-In for Visual Studio .NET*. Retrieved 25/7/06, 2006, from http://www.codeproject.com/dotnet/blogreaderarticle.asp

Crawford, W. (1998). Paper Persists: Why Physical Library Collections Still Matter. *OnLine Magazine*, January/February, pp. 42-48.

Daily, P. D., & Foreman, J. T. (1984). Ada Programming Standards and Guidelines. *Ada Letter*, 3(6), pp. 79-94.

Deimel, L. E. (1991). *Scenes of Software Inspections - Video Dramatizations for the Classroom*. Software Engineering Institute, Carnegie Mellon University.

Department of Computer Science (2006). *Introduction to the Assignment Drop Box*. Retrieved 25/7/06, 2006, from http://www.cs.auckland.ac.nz/tech-support/index.php/Introduction_to_the Assignment_Drop_Box

Dillon, A. (1992). Reading from Paper versus Screens: A Critical Review of the Empirical Literature. *Ergonomics*, 35(10), pp. 1297-1326.

Dix, A., Finaly, J., Abowd, G. D., & Beale, R. (2004). *Human-Computer Interaction (3rd ed.),* London, Pearson/Prentice Hall.

Duda, R., & Hart, P. (1973). *Pattern Classification and Scene Analysis.* New York: John Wiley & Sons.

Dykstra-Erickson, E., & Curbow, D. (1997). The Role of User Studies in the Design of OpenDoc, *Proceedings of the Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques*, Amsterdam, pp. 111-120.

Fagan, M. (1976). Design and Code Inspections to Reduce Errors in Program Development. *IBM System Journal*, 15(3), pp. 182-211.

Freedman, D. P., & Weinberg, G. M. (2000). *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*. New York: Dorset House Publishing Co.

Gilb, T., & Graham, D. (1993). *Software Inspection*. Boston: Addison-Wesley Longman Publishing Co.

Global Graphics (2003). *Jaws PDF Creator, Editor and Server Software*. Retrieved 18/12/03, 2003, from http://www.jawspdf.com

Gocinski, F. (2004). *Tablet 101*. Retrieved 25/7/06, 2006, from http://msdn.microsoft.com/windowsvista/reference/mobilepc/columns

Golovchinsky, G., & Denoue, L. (2002). Moving Markup: Repositioning Freeform Annotations, *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology*, Paris, pp. 21-30.

Green, T. R. G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Approach. *Journal of Visual Languages and Computing*, Vol. 7, pp. 131-174.

Green, T. R. G., Gilmore, D. J., Blumenthal, B. B., Davies, S. P., & Winder, R. (1992). Towards a Cognitive Browser for OOPS. *International Journal of Human-Computer Interaction*, Vol. 4, pp. 1-34.

Heinrich, E., & Lawn, A. (2004). Onscreen Marking Support for Formative Assessment, *Proceedings of the World Conference on Educational Multimedia, Hypermedia & Telecommunications*, Norfolk, pp. 1985-1992.

Huang, A., Doeppner, T. W., & Cetintemel, U. (2003). *Ad-hoc Collaborative Document Annotation on a Tablet PC*. Retrieved 25/7/06, 2006, from http://www.cs.brown.edu/publications/theses/ugrad/2003/ashuang.pdf

Humphrey, W. S. (1989). *Managing the Software Process*. Boston: Addison-Wesley Longman Publishing Co.

iMarkup Solutions (2006). *iMarkup - Business Process Automation, eForms & Reporting*. Retrieved 25/7/06, 2006 from http://www.imarkup.com

InfoWorld (2006). *2006 Technology of the Year Awards: The Winners' List*. Retrieved 25/7/06, 2006, from http://infoworld.com/article/06/01/02/73433_01FEtoyawards_2.html

Janssen, W. C. (2005). ReadUp: A Widget for Reading, *Proceedings of the 9th European Conference ESDL' 05*, Vienna, pp. 230-241.

Jarrett, R., & Su, P. (2002). *Building Tablet PC Applications.* Washington: Microsoft Press.

Joh, F., & Mosley, V. (1989). An Innovative Approach to Software Process Improvement. *IEEE Aerospace and Electronics Conference*, vol.4, pp. 1581-1584.

Johnson, A. (1997). Fax Trends. *Office Equipment and Products*, July, pp. 25.

Johnson, P. M. (1994). An Instrumented Approach to Improving Software Quality through Formal Technical Review, *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, pp. 113-122.

Karat, C., Campbell, R., & Fiegel, T. (1992). Comparison of Empirical Testing and Walkthrough Methods in User Interface Evaluation, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Monterey, California, pp. 397-404.

Knight, J. C., & Myers, E. A. (1991). Phased Inspections and their Implementation. *SIGSOFT Software Engineering Notes*, 16(3), pp. 29-35.

Leveson, N. G., & Turner, C. S. (1993). An Investigation of the Therac-25 accidents. *IEEE Computer Society*, 26(7), pp. 18-41.

Li, Y., Hinckley, K., Guan, Z., & Landay, J. A. (2005). Experimental Analysis of Mode Switching Techniques in Pen-Based User Interfaces, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Portland, Oregon, pp. 461-470.

Lin, J., Newman, M. W., Hong, J. I., & Landay, J. A. (2000). DENIM: Finding a Tighter Fit between Tools and Practice for Web Site Design, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Hague, Netherlands, pp. 510-517.

Lorch Jr., R. F., Lorch, E. P., & Klusewitz, M. A. (1993). College Students' Conditional Knowledge about Reading. *Journal of Educational Psychology*, 85, pp. 239-252.

Manoharan, S. (2006). *Introduction to the C# Language*. Retrieved 25/7/06, 2006, from http://www.cs.auckland.ac.nz/compsci335s2t/lectures/mano/C%23Intro.pdf

Maplesden, D. S. G. (2001). *Tool Support for Design Patterns*, Thesis, The University of Auckland, New Zealand.

Marshall, C. C. (1997). Annotation: From Paper Books to the Digital Library, *Proceedings of the 2nd ACM International Conference on Digital Libraries*, Philadelphia, Pennsylvania, pp. 131-140.

Marshall, C. C. (2003). Reading and Interactivity in the Digital Library: Creating an Experience that Transcends Paper, *Proceedings of the CLIR/Kanazawa Institute of Technology Roundtable*, Kanazawa, 5(4) pp. 1-20.

Marshall, C. C., & Brush, A. J. B. (2002). From Personal to Shared Annotations, *Proceedings of CHI '02*, pp. 812-813.

Marshall, C. C., & Brush, A. J. B. (2004). Exploring the Relationship Between Personal and Public Annotations, *Proceedings of the 4th ACM/IEEE -CS Joint Conference on Digital Libraries*, Tuscon, Arizona, pp.349-357.

McConnell, S. (2000). The Best Influences on Software Engineering. *IEEE Computer Society*, 17(1), pp. 10-17.

McGregor, J. D. (2006). *Component Testing*. Retrieved 25/7/06, 2006, from http://www.cs.clemson.edu/~johnmc/joop/col3/column3.html

Microsoft Download Centre (2006). *Update for Microsoft Windows XP Tablet PC Edition Development Kit 1.7*, Retrieved 25/7/06, 2006, from http://www.microsoft.com/downloads

Microsoft Office Online (2006). *Microsoft Office XP Pack for Tablet PC (Tablet Pack)*. Retrieved 25/7/06, 2006, from http://office.microsoft.com/downloads/

Microsoft Paint (2006). *Windows XP - Microsoft Paint Overview*. Retrieved 25/7/06, 2006, from http://www.microsoft.com/resources/documentation/windows/xp/all/products/en-us/mspaint_overview.mspx

Microsoft PressPass (2000). *Microsoft Delivers First .NET Platform Developer Tools for Building Web Services*. Retrieved 25/7/06, 2006, from http://www.microsoft.com/presspass/press/2000/jul00/pdcdeliverspr.mspx

Microsoft Visual Basic (2006). Visual Basic 6.0 Resource Center. Retrieved 25/7/06, from http://msdn.microsoft.com/vbrun

Microsoft Visual Studio (2005). *Visual Studio 2005*. Retrieved 25/7/06, from http://msdn.microsoft.com/vstudio

Microsoft Windows XP Tablet PC Edition (2005). *What is a Tablet PC?* Retrieved 25/7/06, from http://www.microsoft.com/windowsxp/tabletpc/evaluation/about.mspx

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

Modugno, F., Green, T. R. G., & Myers, B. A. (1994). Visual Programming in a Visual Domain: A Case Study of Cognitive Dimensions, *Proceedings of the 9th Conference on People and Computers HCI '94*, Glasgow, pp. 91-108.

Moran, T. P., Chiu, P., & Van Melle, W. (1997). Pen-Based Interaction Techniques for Organizing Material on an Electronic Whiteboard, *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology*, Banff, Alberta, pp. 45-54.

Myer, R. E (1987). Cognitive Aspects of learning and Using a Programming Language, *Interfacing Thought*, Cambridge: MIT Press, pp. 61-79.

Myers, D., Hargreaves, E., Ryall, J., Thompson, S., Burgess, M., German, D., & Storey, M. (2004). Developing Marking Support within Eclipse, *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology Exchange*, Vancouver, British Columbia, pp. 62-66.

Myers, G. (1978). A Controlled Experiment in Program Testing and Code Walkthrough/Inspection. *Communications of the ACM*, 21(9), pp. 760-768.

Norman, D. A. (1991). Cognitive Artifacts, *Designing Interaction: Psychology at the Human-Computer Interface*, New York: Cambridge University Press, pp. 17-38.

Nunamaker, J. F., Dennis, A. R., Valacich, J. S., Vogel, D. R., & George, J. F. (1991). Electronic Meeting Systems to Support Group Work. *Communications of the ACM*, 34(7), pp. 40-61

Oakhill, J., & Garnham, A. (1988). *Becoming a Skilled Reader*. Oxford: Basil Blackwell.

O'Hara, K. (1996). *Towards a Typology of Reading Goals.* Technical Report EPC-1996-107. Cambridge, UK: Rank Xerox Research Centre.

O'Hara, K., & Sellen, A. (1997). A Comparison of Reading Paper and On-line Documents, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Atlanta, Georgia, pp. 335-342.

Parnas, D. L., & Weiss, D. M. (1985). Active Design Reviews: Principals and Practices, *Proceedings of the 8th International Conference on Software Engineering*, London, pp. 132-136.

Pennington, N. (1987). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs, *Cognitive Psychology*, 19(3), pp. 295-341.

Phelps, T., & Wilensky, R. (2000). Robust Intra-document Locations, *Proceedings of 9th World Wide Web Conference*, Amsterdam, pp. 105-118.

Plimmer, B. E., & Apperley, M. (2003). Freeform: A Tool for Sketching Form Designs, *BHCI* (Vol. 2), Bath, pp. 183-186.

Plimmer, B. E., & Grundy, J. (2005). Beautifying Sketch-Based Design Tool Content: Issues and Experiences, *Proceedings of the Australasian User Interface Conference*, Newcastle, Australia, pp. 31-38.

Plimmer, B., & Mason, P. A. (2006). A Pen-based Paperless Environment for Annotating and Marking Student Assignments, *Proceedings of the 7th Australasian User Interface Conference*, pp. 37-44.

Ramachandran, S., & Kashi, R. (2003). An Architecture for Ink Annotations on Web Documents, *Proceedings of the 17th International Conference on Document Analysis and Recognition*, pp 256-260.

Robillard, M. P., & Murphy, G. C. (2002). Capturing Concern Descriptions During Program Navigation, *Position paper for the 2002 OOPSLA Workshop on Tool Support for Aspect Oriented Software Development, Nov 2002.*

Rosson, M. B., & Carroll, J. M. (2002). *Usability Engineering: Scenario-Based Development of Human-Computer Interaction*. San Francisco: Morgan Kaufmann Publishers.

Rubine, D. (1991). Specifying Gestures by Example, *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques SIGGRAPH '91*, New York, NY, pp. 329-337.

Schilit, B. N., Golovchinsky, G., & Price, M. N. (1998). Beyond Paper: Supporting Active Reading with Free Form Digital Ink Annotations, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Los Angeles, California, pp. 249-256.

Shipman, F. M., Price, M. N., Marshall, C. C., Golovchinsky, G., & Schilit, B. N. (2003). Identifying Useful Passages in Documents Based on Annotation Patterns, *Proceedings of ECDL '03*, Springer Verlag, Heidelberg, pp. 101-112.

Shum, S. (1991). Cognitive Dimensions of Design Rationale, *Proceedings of the 6th Conference on People and Computers HCI '91*, Cambridge, pp. 331-322.

Simon, B., Anderson, R., Hoyer, C., & Su, J. (2004). Preliminary Experiences with a Tablet PC Based System to Support Active Learning in Computer Science Courses, *Proceedings of the 9th Annual SIGCSE Conference on innovation and Technology in Computer Science Education*, Leeds, pp. 213-217.

Software Catalogue (2006). *Extract Icons Soft: Icon Scanner, Web Icons -Professional Icons for Web, Icon Extractor Package*. Retrieved 25/7/06, 2006, from http://www.softplatz.com/software/extract-icons

Stein. R. (1995). *Page versus Pixel*. 1995, from http://www.feedmag.com/95.05dialog1.html

Teasley, B. E. (1994). The Effects of Naming Style and Expertise on Program Comprehension. *International Journal of Human-Computer Studies,* Vol. 40, pp. 757-770.

The Eclipse Foundation (2006). *Eclipse.org Home*. Retrieved 25/7/06, 2006, from http://www.eclipse.org

University of Notre Dame (2005). *Tablet PC Initiative*. Retrieved 25/7/06, 2006, from http://www.nd.edu/~learning/tabletpc

W3C (2005). *W3C Document Object Model.* Retrieved 25/7/06, from http://www.w3.org/dom

W3C (2006). *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. Retrieved 25/7/06, 2006, from http://www.w3.org/tr/rec-xml

Wachsmuth. B. (2003). *SHU Tablet PC Project*. Retrieved 25/7/06, from http://technology.shu.edu/page/shutap+shu+tablet+pc+project!opendocument

Weidenbeck, S. (1991). The Initial Stages of Program Comprehension. *International Journal of Man-Machine Studies*, Vol. 35, pp. 517-540.

Wiegers, K. (2001). When Two Eyes Aren't Enough, *Software Development Magazine*, Oct 2001.

Yahoo! Group News (2006). *Visual Studio.NET Add-ins*. Retrieved 25/7/06, 2006, from http://groups.yahoo.com/group/vsnetaddin

Yang, S., Burnett, M. M., DeKoven, E., & Zloof, M. (1995). *Representation Design Benchmarks: A Design-time Aid for VPL Navigable Static Representations*. Department of Computer Science Technical Report 95-60-3, Oregon State University