Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognize the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form and Deposit Licence.
An Investigation into Freeform, Dynamic, Digital Ink Annotation for Program Code

Craig John Sutherland

A thesis submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy in Computer Science,
The University of Auckland, 2017.
Abstract

Understanding program code is cognitively demanding. One tool that has not been investigated previously is using ink freeform annotations to aid understanding of program code. Before we can investigate using freeform annotations for code comprehension there are some user and technical challenges that need answering.

An iterative research approach was used for this investigation. A single research question was investigated in each iteration and the results informed the next iteration.

In the first iteration, the focus was how and why programmers annotate when understanding program code on paper. The results indicate programmers reading program code use similar types of annotation to those reported for other types of reading. The main reasons for adding annotations were to assist with navigation and offload information from the reader’s memory.

In the second iteration, the focus was how to classify digital annotations; a precursor to other operations. The initial phase used a general purpose automatic recogniser but failed to improve on previously reported results. The second phase combined automatic recognition with user input. While this approach resulted in higher classification accuracy, the accuracy rates were still lower than previously reported for sketch-based recognition. An unexpected finding was the participants did not want to classify all annotations during reading.

In the third iteration, the focus was how to refit (modify) annotations in response to changes in the underlying text. Four classes of annotation were investigated: horizontal lines, vertical lines, enclosures and connectors. Several refitting algorithms were implemented and evaluated for each class of annotation. The findings indicate there are two preferred approaches for refitting annotations: stretching the annotation or splitting it and adding a visualisation showing where the annotation was split.

This thesis makes five main contributions. First, a systematic literature review which provides an overview of the current research. Second, details of how and why programmers annotate code on paper. Third, an implementation of an extensible tool for investigating digital ink annotations on code. Fourth, details on how collaborative intelligence can improve recognition. Fifth, a set of proposals for how to refit annotations based on the annotation classification and user preferences.
For Sammi, Ethan, Alan and Caleb
Acknowledgements

Thank you to the people who have helped me on this PhD journey.

To my wife, Sammi, who was always there for me, both good times and bad. She always encouraged me to persist, even when times were difficult.

To my family, both near and far, for their support and love.

To my main supervisor, Assoc. Prof. Beryl Plimmer, who started me on this journey so many years ago and kept me on track. She has challenged me and supported me throughout this journey and ensured that I finished.

To my co-supervisor, Dr. Andrew Luxton-Reilly, who was always willing to challenge my thinking. His endless reviewing helped improve my writing.

To my fellow students in the HCI research group, who together provided a weekly environment for growth and learning. The membership has changed over the years, here are those who helped most: Paul Schmieder, Adam Roughton, Safurah Abdul Jalil, Rachel Blagojevic, Alex Payne, Sam Kavanagh, Arthur Lewis and Athar Imtiaz.

To the participants in my studies, who gave me their time, and those who helped me find participants. Thanks also to Alan Shum at LeapThought and Gareth Cronin at Wynyard Group, who provided the time and space for me to use their offices.

To Prof. Robert Amor and Dr. Kristin Stock, who provided me opportunities to teach while I studied. This helped to clarify my understanding of HCI and showed new ways of understanding my topic.

To the University of Auckland for providing a Doctoral Scholarship and other grants to support my research and conference attendance.

And finally, to my heavenly Father, who took me from being a failing university student to finishing a PhD. Over the years He has provided the support I needed, both directly and via people, to understand I can do all things through Him.

“Now unto him that is able to do exceeding abundantly above all that we ask or think, according to the power that worketh in us,”

Ephesians 3:20
# Table of Contents

Abstract iii  
Acknowledgements vii  
Table of Contents ix  
List of Figures xiii  
List of Tables xv  
Glossary xvii  

1 Introduction: Annotations, Actually 1 
  1.1 Motivation .................................................. 1  
  1.2 Background ................................................ 5  
  1.3 Research Questions ................................. 8  
  1.4 Overview of Research Approach ............... 9  
  1.5 Contributions ............................................ 11  
  1.6 Publications ............................................. 11  
  1.7 Thesis Structure ........................................ 12  
  1.8 Summary .................................................. 15  

2 Literature Review: The Magnificent Ink 17 
  2.1 Research Method ..................................... 18  
   2.1.1 Research Questions (Step 1) .................. 18  
   2.1.2 Protocol Development and Review (Step 2 and 3) . 19  
   2.1.3 Search Strategy (Step 4) ....................... 19  
   2.1.4 Selection Criteria (Step 5) .................. 21  
   2.1.5 Data Extraction (Step 6) .................... 22  
   2.1.6 Data Synthesis (Step 7) .................... 24
4.3.1 Implementation ........................................... 93
4.3.2 Methodology .............................................. 98
4.3.3 Results ..................................................... 99
4.4 Phase Two ................................................... 101
4.4.1 Implementation ........................................... 101
4.4.2 Additional Functionality ................................ 103
4.4.3 Methodology .............................................. 104
4.4.4 Results ..................................................... 106
4.5 Discussion .................................................... 112
4.6 Summary ....................................................... 116

5 User Study 3: The Curious Case of Annotation Adaptation 117
5.1 Introduction .................................................. 118
5.2 The Approach ............................................... 119
5.3 Implementation .............................................. 119
  5.3.1 Horizontal Lines ........................................ 120
  5.3.2 Vertical Lines and Enclosures .......................... 121
  5.3.3 Connectors .............................................. 123
5.4 User Study .................................................... 124
  5.4.1 Procedure ............................................... 125
  5.4.2 Analysis ................................................ 126
5.5 Results ....................................................... 127
  5.5.1 Round One .............................................. 127
  5.5.2 Round Two .............................................. 130
5.6 Discussion .................................................... 135
5.7 Summary ....................................................... 137

6 Discussion: Back to the Annotation 139
6.1 The How and Why of Programmer Annotation .......... 140
  6.1.1 The Two Taxonomies of Annotations ................. 140
  6.1.2 Annotations for Navigation ............................. 142
  6.1.3 Annotations for Offloading Information ............ 143
6.2 Annotations in a Digital Environment ....................... 146
  6.2.1 Stringing Operations Together ........................ 146
  6.2.2 When is Recognition Needed? ......................... 149
  6.2.3 Changing the Appearance of Annotations .......... 151
  6.2.4 Implementing Freeform Annotations ................ 152
6.3 Summary ....................................................... 157
# 7 Conclusions: The Day the Pen Stood Still

## 7.1 Research Questions Revisited

## 7.2 Contributions

## 7.3 Future Work

## 7.4 Conclusions

# A Ethics approval letters

# B vsInk Architecture

### B.1 Basic Architecture

#### B.1.1 Data model

#### B.1.2 UI Layers

#### B.1.3 Processing Digital Ink

#### B.1.4 Rendering annotations

### B.2 Segmenting and Anchoring Annotations

### B.3 Rendering Annotations

### B.4 Adornments

# C Interfaces for Extending vsInk

### C.1 IRecogniser

### C.2 IStrokeTypeProcessor

### C.3 IMergableStrokeTypeProcessor

### C.4 StrokeRendererBase

### C.5 AdornmentGeneratorBase

# Bibliography
# List of Figures

1.1 Examples of text annotations. ........................................... 3
1.2 Iterative research approach used in this thesis. .................. 10

2.1 Publications included in the corpus by year. ....................... 29
2.2 Percentage of implementations for each input mechanism. ...... 30
2.3 Number of implementations that handle each document format. 33
2.4 Number of implementations that recognise each category of annotations. ................................................................. 39

3.1 Annotation types added on the source by classification per participant. ................................................................. 70
3.2 Annotations added to each location per participant. ............ 71
3.3 Examples of annotations used for navigation. ...................... 73
3.4 Examples of annotations used for *working information* and *sharing information*. .................................................. 75
3.5 Example of an unknown annotation. .................................. 78
3.6 Examples of annotations on separate pages. ....................... 80
3.7 Examples of spread out stacks of paper. ......................... 81

4.1 *vsInk* running inside *Visual Studio 2013*. ..................... 93
4.2 Calculation of context overlap. ...................................... 95
4.3 Different feature functions. .......................................... 97
4.4 Screenshot of verification approach. ............................. 102
4.5 Screenshot of constraint approach. ............................... 103
4.6 Participants’ ratings about the general experience. ............ 108
4.7 Participants’ ratings for each classification approach. .......... 109

5.1 Examples of Horizontal Line Refitting. .......................... 120
5.2 Examples of Vertical Line and Enclosure Refitting. ............ 121
5.3 Examples of Connector Refitting. ................................ 123
# List of Tables

1.1 Classification of annotation by dimension. .................................. 5

2.1 Data extraction form. .......................................................... 22
2.2 Publications included in the mapping study. ................................. 26
2.2 (continued). ........................................................................ 27
2.2 (continued). ........................................................................ 28
2.3 Initial Taxonomy of Annotation Types. ........................................ 36
2.4 Taxonomy of Annotation Support Operations. ............................... 37
2.5 Annotation category recognised by implementation. ................. 38
2.6 Examples of each annotation type. ........................................... 40
2.6 Continued. ............................................................................ 41
2.7 Number of implementations for each anchoring approach. .......... 45
2.8 Number of implementations that automatically adapt annotations. 46
2.9 Adding and Adapting operations by implementation. .................. 48
2.9 (continued). ........................................................................... 49

3.1 Background Information on Participants. ................................. 68
3.2 Annotations by Participants. .................................................... 69
3.3 List of Reasons for Adding Annotations. .................................... 72

4.1 Contextual Features for Classifying Strokes. .............................. 95
4.2 Heuristic Features for Classifying Strokes. .................................. 96
4.3 Formulas for Classifying Strokes. ............................................. 98
4.4 Number of annotations collected for evaluating the classifiers. .... 100
4.5 Accuracy of each recognizer by annotation class. ....................... 101
4.6 Categories of annotations added. ........................................... 106
4.7 Accuracy Rates for each Approach. ......................................... 106
4.8 Annotations erased by approach. .......................................... 108
4.9 Participants’ preferences for classification approach. ................. 110
Glossary

Annotation a semantic grouping of ink strokes into single unit.
Enclosure a circular annotation enclosing underlying content.
Orphaning removing an annotation in response to the underlying content being deleted. This occurs when there is insufficient context remaining to reposition or refit the annotation.
Refitting changing the appearance of the annotation in response to a change in the underlying content.
Stroke a series of points drawn between pen-down and pen-up.
Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Parts of Chapter 2 are extracted from:


<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>All work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>95</td>
</tr>
</tbody>
</table>

**CO-AUTHORS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beryl Plimmer</td>
<td>Supervision, editing</td>
</tr>
<tr>
<td>Andrew Luxton-Reilly</td>
<td>Supervision, editing</td>
</tr>
</tbody>
</table>

**Certification by Co-Authors**

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and

- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beryl Plimmer</td>
<td></td>
<td>28/1/2016</td>
</tr>
<tr>
<td>Andrew Luxton-Reilly</td>
<td></td>
<td>28/07/2016</td>
</tr>
</tbody>
</table>

Last updated: 19 October 2015
This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Parts of Chapter 4 are extracted from:


<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>All work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>95</td>
</tr>
</tbody>
</table>

## CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beryl Plimmer</td>
<td>Supervision, editing</td>
</tr>
<tr>
<td>Andrew Luxton-Reilly</td>
<td>Supervision, editing</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beryl Plimmer</td>
<td></td>
<td>28/07/2016</td>
</tr>
<tr>
<td>Andrew Luxton-Reilly</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Parts of Chapter 5 are extracted from:


<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>All work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>95</td>
</tr>
</tbody>
</table>

### CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beryl Plimmer</td>
<td>Supervision, editing</td>
</tr>
<tr>
<td>Andrew Luxton-Reilly</td>
<td>Supervision, editing</td>
</tr>
</tbody>
</table>

### Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beryl Plimmer</td>
<td>[Signature]</td>
<td>28/7/2016</td>
</tr>
<tr>
<td>Andrew Luxton-Reilly</td>
<td>[Signature]</td>
<td>28/07/2016</td>
</tr>
</tbody>
</table>
This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Parts of Chapter 6 are extracted from:


<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>All work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>95</td>
</tr>
</tbody>
</table>

**CO-AUTHORS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beryl Plimmer</td>
<td>Supervision, editing</td>
</tr>
<tr>
<td>Andrew Luxton-Reilly</td>
<td>Supervision, editing</td>
</tr>
</tbody>
</table>

**Certification by Co-Authors**

The undersigned hereby certify that:
- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beryl Plimmer</td>
<td><img src="signature_image" alt="Signature" /></td>
<td>25/7/2016</td>
</tr>
<tr>
<td>Andrew Luxton-Reilly</td>
<td><img src="signature_image" alt="Signature" /></td>
<td>28/07/2016</td>
</tr>
</tbody>
</table>

Last updated: 19 October 2015
CHAPTER 1

Introduction
Annotations, Actually

“To raise new questions, new possibilities, to regard old problems from a new angle, requires creative imagination and marks real advance in science.”

Albert Einstein

This chapter introduces the why and how of this thesis and provides an overview of the remainder of the thesis. Section 1.1 explains why this research is important and how it fits in with other research in this space. Section 1.2 provides the context and theory for the remainder of the thesis. This is followed in Section 1.3 with the main research objective and questions. Section 1.4 outlines the research approach followed to address these questions. Section 1.5 then lists the main contributions from this thesis. Section 1.6 lists the papers that were accepted for publication during the course of the thesis. Finally, section 1.7 outlines the structure of the rest of the thesis and explains how each part contributes to the research objective.

1.1 Motivation

Handwritten annotations are an easy and effective way to actively engage with a document (Adler and Van Doren, 1972, O’Hara and Sellen, 1997) that have been shown to improve comprehension and retention (Simpson and Nist, 1990, Ball et al., 2009). Modern pen and touch input devices allow for freeform digital ink annotations similar to pen on paper (Morris et al., 2007). There are
numerous approaches that support digital ink annotations on static documents but annotating dynamic documents poses significant technical challenges that remain unsolved. Yet a core advantage of most digital document formats is their inherent support for change.

The impetus for ink annotation support in digital tools is that freeform ink annotations offer two benefits over text annotations. First, annotating with a pen is less cognitively demanding than with a mouse and keyboard. Second, ink annotations stand out from the underlying text and are easier to find (O’Hara and Sellen, 1997).

Adding annotations using a keyboard and mouse requires a higher mental workload than using a pen. The user has to switch from thinking about what they are reading to how they are going to annotate (O’Hara and Sellen, 1997). This mental switch is increased by software implementations that require the reader to first select text to anchor the annotation to and then add the annotation. Consequently, people annotate less on a computer screen than on paper (Liu, 2005). Also, the increased mental workload of text annotation reduces how much the reader can comprehend and learn (Kawase et al., 2009).

As well as being more cognitively demanding, it is harder to find text annotations than freeform ink annotations (O’Hara and Sellen, 1997). One important role of annotations is to act as signposts to important information (Fowler and Barker, 1974). Considering Figure 1.1, the ink annotations stand out from the page making them easier to see when scanning through a document. In contrast, even with colour coding the text annotations blend with the text forcing the reader to spend more time looking for them and less time on comprehension.

While the benefits of freeform ink annotations are recognised (Schilit et al., 1998a, Morris et al., 2007, Kawase et al., 2009) there are technical challenges involved in adding freeform ink annotations to digital text documents. Computers can process the data in text documents due to the structure of the documents. A document consists of a string of characters which are grouped into words, sentences, paragraphs and so on. In contrast, freeform ink annotations lack this intrinsic structure: it is more difficult to extract the meaning contained within it. While it is possible to treat annotations as images on a document, this limits functionality available. Instead, an application needs some way to unlock the meaning within an annotation.

Substantive documents (e.g. academic papers, legal contracts) can go through
multiple drafts. While reviewing a draft, annotations are used to add questions and suggestions, correct errors and so forth (Adler et al., 1998, Sellen and Harper, 2002). Then the draft is updated, resulting in changes to the content and/or structure. With modern WYSIWYG editors, it is possible to generate, annotate and edit the document all within one environment. Tools such as Microsoft Word even allow adding freeform ink annotations. While using a single tool for the entire editing process simplifies it, the question is: How should ink annotations adapt as the underlying document changes?

Instead of using a WYSIWYG editor, it is possible to generate documents from tools like \LaTeXX. These tools generate static documents, such as PDFs, that are more difficult to edit. While there are tools that allow freeform ink annotations on static documents (Price et al., 1998b, Chen et al., 2012), there are two challenges. First, the person editing the document needs to mentally translate the location of the annotations on the document to the location of the source for the document. Second, as the document is being regenerated every time, tracking the history of the document is more difficult.

Another approach is to convert a document to a static image\(^1\). Annotations are

\(^1\)A manual example of this process is printing the document onto paper.
then positioned using standard Cartesian coordinates (Schilit et al., 1998a, Olsen et al., 2004). Again, this simplifies the process of adding annotations with the same issues as annotating on a static document.

One domain where documents are frequently edited is computer programming. Program code documents are regularly changed throughout their lifetime; often by different people. At the same time, the characteristics of program code make it more difficult to understand. Thus, it would be beneficial to incorporate freeform ink annotations as a tool to help comprehension without incurring the issues of annotating on static documents or images.

Some of the characteristics that impact code comprehension are the size and complexity of the computer programs. Reading through the code does not typically follow linear paths from start to end but instead has complex, nonlinear paths. This makes program code difficult to read through as the reader needs to remember what has happened beforehand and slot it together with what they are currently reading. In addition to non-linearity, programs grow and evolve over time. This typically results in increasing amounts of code in the program. At the same time, old code is often changed. Thus, as the size of the program increases it becomes less feasible to print out and read the code on paper; especially when the programmer does not know what path through the code they need to follow.

When changing a computer program, programmers need to understand a computer program first. Typically, the first activity undertaken is to read through what they think are the relevant sections of the code to understand it (LaToza et al., 2007, Sillito et al., 2008). With large programs, programmers search through the code to find the relevant sections. Only when they understand the code will they change it. Modern Integrated Development Environments contain tools to help with this process but only a small number of tools in modern Integrated Development Environments are as intuitive and easy to use as freeform ink annotations (Sillito et al., 2008).

Most people already annotate using pen and paper; therefore it would be easy for programmers to learn an ink annotation tool; especially one that is based on natural freeform annotation practices. Thus, annotations are potentially of benefit to programmers while reading for comprehension.


### Table 1.1: Classification of annotation by dimension, based on Marshall (1997, p135).

<table>
<thead>
<tr>
<th></th>
<th>Within-text</th>
<th>Marginal or blank space</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Telegraphic/Tacit</strong></td>
<td>Underlining; Highlighting; Circles and enclosures around words</td>
<td>Brackets and braces; Asterisks and stars; Circles and enclosures around whole pages; Arrows and connectors connecting within-text and marginal markings</td>
</tr>
<tr>
<td><strong>Explicit</strong></td>
<td>Brief notes between lines</td>
<td>Short phrases in margin Extended notes in margin Extended notes on blank pages Problems worked in margins</td>
</tr>
</tbody>
</table>

### 1.2 Background

When classified by form and appearance, annotations can be classified along two dimensions\(^2\): whether the annotation was within the text or margin, and whether the annotation was telegraphic or explicit in meaning (Marshall, 1997). Based on these two dimensions, Marshall was able to classify several ‘classes’ of annotations. These are listed in Table 1.1.

Initially, Marshall (1997, p136) defined six categories of reasons why the reader may have added the annotation\(^3\):

- Procedural signalling for future attention;
- Placemarking and aiding memory;
- Tracing progress through difficult narrative;
- In place problem-working;
- Interpretation of the text;
- Incidental reflection of the material circumstances.

\(^2\)There is also a third dimension: whether the annotation was removable. However, Marshall (1997, p134) then excluded it from the analysis as Marshall was only interested in permanent annotations.

\(^3\)This is deliberately vague as Marshall did not interview the annotators; instead, classifying them based on the content and location of the annotations.
Procedural signalling for future attention annotations are added in anticipation of future use: they show something that the annotator thinks they will return to later. These may be to mark something as important or to reduce future rereading time (Marshall, 1997). These annotations are therefore a form of navigation aid (Kawase et al., 2009).

In contrast, placemarking and aiding memory annotations are intended to help with the process of memorisation. For example, highlighting key words or phrases in a document (Marshall, 1997). Thus, these annotations function as an alternate form of navigation for quickly finding these terms again.

Finally, Tracing progress through difficult narrative annotations are a third form of navigation markings. These markings provide a visible trace of what the reader has already read (Marshall, 1997). This allows the person to quickly return to where they were previously. In addition, these annotations help maintain the reader’s attention, especially for dense texts, thus aiding reading.

In place problem-working annotations also provide benefits by allowing the reader to work out a problem in context. This allows the reader to combine information about the problem with the problem itself (Jackel, 2014). In addition, it reduces the risk of a break in attention caused by a shift to a different medium (Marshall, 1997).

Interpretation of the text annotations are also a form of in-text working. These annotations record what the reader understands about the text (Marshall, 1997).

The last two classifications are both potential examples of external cognition (Scaife and Rogers, 1996a,b, Rogers, 2004). This is where the reader is using the text to offload some form of cognition to the paper to reduce their mental load.

The final classification, incidental reflection of the material circumstances, refers to annotations that have no relationship to the text itself. These may be due to external distractions or due to the availability of the paper in the book (Marshall, 1997). These annotations provide less long benefits to the reader.

Marshall (1997) also posited there is an interplay between form and function. For example, procedural signalling annotations tend to be telegraphic annotations, either within the text (e.g. underlines) or in the margin (e.g asterisks). Thus, it may be possible to classify the function of the annotation based on its form, assuming that it is possible to accurately classify the annotation based on its form. While these classifications cover a wide variety of annotations, there are some annotations that are classified by one scheme and not the other. For
example, doodles and drawings are classified in the functional scheme but not the form scheme.

In the second study on annotations, Marshall expanded the reasons why annotations were added and defined a number of dimensions that annotations can be classified by (Marshall, 1998). Of relevance to this thesis are the following dimensions:

**Formal vs. informal** refers to whether the annotation style follows a standard or not (Marshall, 1998). While some industries define a standard set of symbols for annotations (e.g. copy editing and exam marking) programmers do not have any such standards. Some Integrated Development Environments provide conventions for some types of text comments (e.g. prepending TODO to a comment or comments used for generating automated documentation) but most annotations are informal (Storey et al., 2009).

**Explicit vs. tacit** refers to how understandable an annotation is to another reader (Marshall, 1998). By default, reading code in an electronic environment limits the reader to textual comments which tend to be explicit; however Marshall posits that readers who are immersed in the text would tend towards tacit (telegraphic) annotations.

**Permanent vs. transient** refers to how long annotations are kept (Marshall, 1998). Some annotations are transient, added as the reader moves through the text. These annotations have no value beyond that reading session. In contrast, some annotations are intended for long-term use, even to the point of having value for future readers (Marshall, 1998, Agosti et al., 2007). Most annotations added on program code tend to be permanent. This is due to the effort needed to add them and their textual nature (Storey et al., 2009).

**Public vs. private** refers to the intended audience of the annotations. Private annotations are intended only for the reader (and sometimes only for that reading session), whereas public annotations are intended to be shared (Marshall, 1998). Once again, there is a bias for code-based annotations in current environments. For this dimension the annotations tend to be public annotations. Again, this is due to limitations of how code can currently be annotated (Storey et al., 2009).

---

4The reader is referred to Marshall (1998) for the additional three dimensions.

5Especially as there are some schools of thought that state comments should be added to help future programmers.
One key characteristic of most annotations is they are idiosyncratic. Every person has their own style of annotations, and even these styles tend to change over time (Marshall, 1997, 1998). Even a formal style of annotations depends on how the person uses them. The fluid nature of annotations makes it hard to categorise annotations, as even annotations with a similar form can be used for different functions.

One form of annotation that was not included in Marshall’s early work is collaborative annotations. Collaborative annotations are annotations intended for use amongst a group of people as a way of sharing information (Marshall and Brush, 2004). This was due to Marshall studying textbooks rather than people (Marshall, 1997). While collaborative annotations can be classified using the dimensions used above, they have a different profile. These differences also affect the form and function of the annotations. For example, underlines are the most common private annotation while text is the most common public annotation (Marshall and Brush, 2004).

Annotations added for personal use tend to be tacit and informal. This allows the person to focus on the reading task rather than the annotations themselves (Kawase et al., 2009). As such, they are more an aid to navigation than understanding. In contrast, public annotations tend to be either more explicit or more formal. This allows sharing understanding about the annotations and what they mean (Marshall and Brush, 2004). While annotations may start off as private, they may go through a transformation when moved to the public realm.

This section shows how Marshall’s (1997) work forms a basis for classifying annotations. This work, together with further research on collaborative annotations, provides insight into why people annotate. With this background, the next section changes focus to the research questions for this thesis.

1.3 Research Questions

The main objective of this thesis is to investigate “How we can support freeform digital ink annotations for programming in an Integrated Development Environment.” To achieve this objective, the following research questions were posed:

RQ1: How and why do programmers annotate programming code on paper?
Chapter 1. Introduction

RQ2: How can an Integrated Development Environment be extended to support annotation of digital dynamic code documents in a natural way?
RQ2.1: How can the recognition of annotations be improved?
RQ2.2: How can annotations be automatically refitted in response to changes in the code?

The first research question (RQ1) addresses how programmers would annotate program code with freeform ink. This aims to discover what types of annotations a programmer would add. The type of annotation is defined either by the reason the annotation was added or by the characteristics of its appearance. Both categories are needed to implement an annotation system. As there are currently no annotation systems in an Integrated Development Environment this was investigated by observing programmers as they read the code on paper.

The second research question (RQ2) addresses how to overcome some of the technical and human challenges needed to implement an annotation system. This is important as there are many operations needed to add and adapt annotations within dynamic documents. While some of the operations have proven solutions, others have not been investigated. As the literature review will show, there are two major unsolved related technical issues. Each of these issues is addressed in a sub-question. The first sub-question (RQ2.1) is: “How can we improve the recognition process when new freeform ink annotations are added to a document?” The answers to this question then feed into the second sub-question (RQ2.2): “How can we automatically adapt annotations in response to changes in the underlying document?”

1.4 Overview of Research Approach

Figure 1.2 shows a high-level overview of my research approach. This is based on the “The Research Cycle” in Leedy and Ormrod (2010, p. 8).

The approach starts off with the formulation of a problem (1). This is typically in the form of one or more questions to investigate. Once the problem has been formulated, one or more hypotheses are posited as solutions to the problem (2). The researcher then collects data to evaluate the hypotheses (3). The data is then analysed and interpreted to assess how well the hypotheses resolve the initial problem (4). This interpretation provides new information added to what is already known, which is then communicated to the wider field (5). Based on
the new information, the problem is reformulated and new questions posed (6). These new questions can start new cycles of the investigation.

This thesis explores the problem: “How can we support freeform digital ink annotations for programming in an Integrated Development Environment?” The literature review identified some studies that implemented freeform digital ink annotations in an Integrated Development Environment, however there are no studies addressing programmers’ current annotation practices. Therefore, to answer this question, we first need to know if and how programmers currently use annotations. As there are currently no commercial implementations of freeform digital ink annotation in an Integrated Development Environment, paper was used for exploring how and why programmers annotate. This study was used to inform the development of tools to support annotation in Integrated Development Environment. This is formulated as **RQ1**: “How and why do

---

6The implementations reported in Chapter 2 are only proof-of-concept research implementations.
programmers annotate programming code?" This question was addressed by a user study into how experienced programmers annotate program code on paper. The methodology for this study is in Chapter 3.

The results of this study showed that programmers might still annotate if they had the tools to do so. To address this deficiency a framework was implemented in Visual Studio 2013 to explore possible solutions. This framework, vsInk, is described in more detail in Appendix B, being the primary artefact for investigating the reformulated problems (Peffers et al., 2007).

After the first iteration, the problem was reformulated as **RQ2.1**: "How can the development environment be extended to support freeform ink annotations?" From the systematic literature review in Chapter 2, two additional areas were chosen for investigation: recognising annotations and refitting them. As recognising the annotation class is an important pre-requisite for refitting this was selected as the next area of investigation. The problem was reformulated as: "How can we improve the recognition of freeform ink annotations on digital text?" This problem was explored in two phases. These phases, and the associated user study, are in Chapter 4.

Using the new information from the previous iteration, the next question for investigation was reformulated as **RQ2.2**: "How can freeform annotations be automatically refitted in response to changes in a dynamic digital document?" A series of potential solutions were implemented in vsInk and evaluated in a final user study. The details of this study are in Chapter 5.

### 1.5 Contributions

This thesis makes the following contributions:

- A systematic overview of freeform digital ink annotations, identifying key operations for adding and adapting annotations and an overview of the published research (Chapter 2);
- A taxonomy of how and why programmers annotate on paper and how these may assist code comprehension (Chapter 3);
- An extensible platform (vsInk) for investigating freeform digital ink annotations on program code (Appendix B);
- Recommendations on how the classification may be improved, especially via the use of collaborative intelligence (Chapter 4);
• And a set of research-backed proposals on how to refit annotations on dynamic documents (Chapter 5).

1.6 Publications from This Thesis

The following publications were produced from research conducted during the thesis. Some of the content from chapters in this thesis are based on the publications below:


CJ Sutherland, A Luxton-Reilly, and B Plimmer. Who changed my annotation? an investigation into refitting freeform ink annotations. In Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on, VL/HCC ’16, 2016b

CJ Sutherland, A Luxton-Reilly, and B Plimmer. Location, location, location: Using spatial memory in an IDE to assist program code comprehension (work in progress). In Psychology of Programming 2016, PPiG, 2016c

1.7 Thesis Structure

This thesis investigates how freeform digital ink annotations could be supported in an Integrated Development Environment. Chapter 1 provides the motivation and some background on freeform ink annotations. Chapter 2 reviews the prior literature in this area. Chapters 3 to 5 present the three iterations in this thesis, including their results and analysis. Finally, Chapter 6 discusses the implications of this work and links it to the broader research in digital ink annotations while
Chapter 7 summarises the contributions and suggests future directions. Each chapter is described in more detail below.

**Chapter 1: Introduction.** This chapter introduces my research and provides an overview of the thesis. It starts with the motivation for the research, followed by background on annotations and the research questions. This is followed by the contributions of the thesis and a list of publications generated from it. Finally, the chapter finishes with an outline of the thesis organisation.

**Chapter 2: Literature Review.** The second chapter describes a systematic mapping study performed to determine the current state of research. A systematic mapping study is a repeatable approach for generating an overview of the literature in an area. The focus of this study was how to add and adapt freeform digital ink annotations on dynamic documents. A total of 63 publications were included; these describe 43 different implementations.

The chapter describes five aspects of freeform digital ink annotations. First, where annotations have previously been implemented (domain, format and input mechanisms used)? Second, the types of annotations that have been investigated previously. Third, the operations needed for adding and automatically adapting annotations, and what influence the annotation type has on these operations. Fourth, how annotations have been previously implemented in Integrated Development Environments. Finally, how the user expectations of these annotations have been investigated previously.

Together, the aspects provide a picture of how freeform digital ink annotations have previously been implemented and what is currently in the published research. Three areas were identified for investigation: whether programmers would want to annotate, and if so, how; how to accurately classify the type of annotation; and how to automatically adapt annotations when the content changes. The remainder of my research investigates these three areas.

**Chapter 3: User Study 1.** The fourth chapter investigates whether programmers would want to annotate, and if so, how? This investigation was performed using an observational user study. Experienced programmers read a short program on paper for understanding. Of interest was whether
the programmers added any annotations, what types of annotations and why they added them.

In general, the participants made some form of marks, either as annotations on the code or on separate note pages. The types of annotations added were similar to other reported types of annotations on books. In addition, the main reason programmers annotated was as a form of navigation; information offloading was the other common reason. This suggests that freeform annotation may still be beneficial for programmers.

**Chapter 4: User Study 2.** The fifth chapter changes focus to the technical challenges of implementing freeform digital ink annotations. This chapter focuses on how to accurately classify the annotation type automatically, using a computer-based recogniser. Knowing the annotation category is an important precursor for adapting annotations. Previous work in the sketching domain has reported high accuracy rates but it is unknown whether the same approaches would work as well in an annotation context.

In the first phase of this study, different algorithmic approaches were implemented for automatically classifying the annotation types. The best approach was then combined with two user-inclusive approaches and evaluated in a within-subject usability study. The participants in this study annotated several code files using the new approaches. One key result is none of the approaches (automatic or user-inclusive) were as accurate in an annotation domain as in freeform sketching. However, the results do suggest some avenues of future investigation for improving classification.

**Chapter 5: User Study 3.** The third study addresses how to automatically adapt freeform ink annotations? This chapter specifically focuses on refitting annotations. Several different approaches were implemented and evaluated using a within-subject usability study.

In this study, the participants were asked to add annotations of different types to a program code file. The underlying code was modified and the annotations refitted in response. The participants then rated each approach for how well it maintained the original meaning. The results suggest the preferred ways of refitting four classes of annotation: horizontal lines, vertical lines, enclosures and connectors.

**Chapter 6: Discussion.** The seventh chapter ties together the different threads of this thesis in addressing the main research objective: “How we can
support freeform digital ink annotations for programming in an Integrated Development Environment.” This chapter discusses several ways this thesis advances the knowledge in this area.

**Chapter 7: Conclusions.** The final chapter in this thesis summarises the main contributions and their significance. It also suggests some future lines of investigation for the main research topic.

In addition to these chapters, there are also three appendices. The first appendix includes ethics approvals for the studies in this thesis. The second appendix explains how the main research tool, *vsInk*, was implemented. The final appendix provides some documentation on the main extension points of *vsInk*.

### 1.8 Summary

This chapter has provided the motivation and context for investigating freeform digital ink annotations for program code. The next chapter starts with exploring the prior research on implementing freeform digital ink annotations on dynamic documents.
CHAPTER 2

Literature Review
The Magnificent Ink

“Nothing has such power to broaden the mind as the ability to investigate systematically and truly all that comes under thy observation in life.”

Marcus Aurelius

The previous chapter introduced the thesis and provided a high-level overview of the background research. This chapter reviews the previous research on dynamic, freeform digital ink annotations. It contains the results of a systematic mapping study carried out to survey the research on digital ink annotations. The purpose of this mapping study was to identify what has been investigated previously for freeform digital ink annotations; both for Integrated Development Environments and in general. This, in turn, allows us to identify what the currently published research does not cover in order to direct the research for the remainder of the thesis.

Section 2.1 explains the methodology used to perform the literature review. This includes the detailed steps of the protocol and how it was developed. Next, section 2.2 contains the results of the literature review, including the resulting taxonomies developed. Section 2.3 discusses these results and associates them with the main research goals of the thesis. Finally, section 2.3 summarizes the key points.

Some of the results from this systematic mapping study have been published previously in Sutherland et al. (2016a).
2.1 Research Method

Systematic literature reviews originated in the field of medical studies. They are used to perform a systematic, comprehensive and reproducible analysis of the research about a given topic. They have been applied in other fields such as software engineering, social sciences, chemistry and education (Kitchenham and Charters, 2007). A systematic mapping study is a variation that provides a wider overview of a research area. They are useful for identifying what has and has not been researched (Kitchenham and Charters, 2007, Okoli and Schabram, 2010). The attributes of this protocol make it ideally suited to the purposes of documenting and synthesising research on digital ink annotation.

This study used the protocol described by Okoli and Schabram (2010), which involves the following steps:

1. Specify the research questions
2. Protocol development
3. Protocol review
4. Study search
5. Primary studies selection
6. Data extraction
7. Data synthesis

As this is a systematic mapping study, rather than a systematic literature review, there was no analysis of quality (Kitchenham et al., 2010).

Each step is fully documented to ensure that the study can be reproduced.

2.1.1 Research Questions (Step 1)

Due to the nature of RQ2, (“How can an integrated development environment be extended to support annotation of digital dynamic documents in a natural way?”) this study primarily focuses on how annotations have been implemented. The technical focus was chosen to investigate how annotations could be implemented (e.g. how the user interface was implemented, what algorithms and technology was used, etc.) The purpose of focusing on prior technical implementations was to inform how digital freeform ink annotations could be implemented in Visual Studio.

The questions informing this mapping study are:
MS.Q1: What operations are needed for adding and adapting annotations on dynamic, digital documents?

MS.Q2: What types of annotations need to be handled in an annotation application?

MS.Q3: What are the user expectations for annotation adaptation?

MS.Q4: What has been implemented for Integrated Development Environment integration in the area of freeform annotations?

2.1.2 Protocol Development and Review (Step 2 and 3)

Five publications from a previous study (Sutherland and Plimmer, 2013) were used to identify potential key terms. These terms were used to define the primary search string and possible alternatives.

A data extraction form was also developed. This form listed the data items to obtain from each publication. These items were chosen based on the research questions. Some of these items are selections with the initial values based on values from the initial five publications.

During the development of the protocol, the data extraction form was trialed in a small group. The trial evaluated the definitions of each item to ensure consistency. One of the publications from the previous study was used (Golovchinsky and Denoue, 2002). Each member extracted all the data items. Based on the feedback the data extraction form was modified to clarify the definitions of each item. The final form is shown in Table 2.1.

2.1.3 Search Strategy (Step 4)

The search string, Annotation AND “Digital Ink”, was trialled on the following six databases:

1. ACM Digital Library
2. IEEE Xplore
3. SpringerLink
4. Scopus
5. Inspec
6. ProQuest
The search string found all five of the initial publications. Some alternate search strings were also tried but these either did not add any additional results or were too broad.

After the search string was finalized the search was run on all six databases on a single day. The databases were searched in the order listed above. Where possible a full-text search was used; otherwise, the fullest search options were used.

During the search, the results were extracted into a table. The information recorded in the table for each publication included:

(i) Year of publication
(ii) Venue
(iii) Authors
(iv) Title
(v) Source

Duplicate results from multiple databases were added to the table. When there were duplicate results within a database (e.g. a conference proceeding and journal article for the same study) the most recent one was used.

After the initial set of publications was selected (Step 5 - see §2.1.4) a forward and backward search was performed using the references and citations of each selected publication. After this search, Step 5 was re-applied to the new results. Only one iteration of forward and backward searching was performed.

Finally, both a new database search and a new forward and backward search were run to detect any new research published after the initial search. These searches were limited to publications in 2014 or later.

In total, three searches were run. The initial search was performed on 24 December 2013. The backwards and forwards search was performed on 18 March 2015. The results from these two initial searches were published in Sutherland et al. (2016a). A final search, including both database and forward and backward searches, was performed on 19 May 2016. This final search was used to update the results and includes studies published in 2014 and 2015\textsuperscript{1}.

\textsuperscript{1}There were no studies published in 2016 that match the search criteria.


2.1.4 Selection Criteria (Step 5)

The final publications were selected using multiple phases. In the first phase, the source details of each publication were checked. Non-peer reviewed publications (e.g. magazine articles) and non-English publications were excluded. Duplicates based on the authors, date and title were also identified and excluded: only the most recent publication was included in the results.

In the second phase, publications whose topic was out of scope were excluded as the focus of this review was specifically the annotation of text-based documents. Therefore, publications examining annotation of video and audio files were excluded as they are not text-based (e.g. Cabral and Correia, 2009). Whiteboard applications were excluded for the same reason (e.g. Brandl et al., 2008). Drawing and sketching applications were excluded as they start with a blank canvas rather than a pre-existing document (e.g. Zeleznik et al., 2008). Finally, implementations with text-only annotations were excluded as the focus was specifically freeform digital ink (e.g. Zyto et al., 2012). The exclusion criteria were applied using the publication’s title and abstract. If there was any doubt about whether a publication should be excluded, it was left in. When a publication was excluded, the reason for exclusion was noted in the table.

For the next phase, the full-text of each remaining publication was retrieved. If a publication could not be retrieved in this phase, it was excluded. An example of a publication that could not be retrieved is one that is marked as a citation in the search engine without a link or DOI to retrieve it. With these publications, all possible attempts to retrieve them were made; including using multiple libraries and other search engines.

For the final phase, the abstract and conclusion of each publication were checked to ensure it met the following inclusion criteria:

(i) The publication must include an implementation of a system;
(ii) The implementation must allow users to add annotations;
(iii) The implementation must use digital ink;
(iv) The annotations must be for a document (e.g. not a blank notebook).

If there was insufficient detail in the abstract and conclusion to determine whether to include the publication, the rest of the publication was scanned. Any publication that did not meet the inclusion criteria was excluded.
The Magnificent Ink

Table 2.1: Data extraction form.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Data</td>
<td></td>
</tr>
<tr>
<td>Year of Publication</td>
<td>Numeric</td>
</tr>
<tr>
<td>Authors</td>
<td>Free text</td>
</tr>
<tr>
<td>Title</td>
<td>Free text</td>
</tr>
<tr>
<td>Publication venue (e.g. conference or journal name)</td>
<td>Free text</td>
</tr>
<tr>
<td>Abstract</td>
<td>Free text</td>
</tr>
<tr>
<td>Name of implementation</td>
<td>Free text</td>
</tr>
<tr>
<td>System Data</td>
<td></td>
</tr>
<tr>
<td>Overview</td>
<td>Free text</td>
</tr>
<tr>
<td>Input mechanism</td>
<td>Selection</td>
</tr>
<tr>
<td>Application Domain</td>
<td>Free text</td>
</tr>
<tr>
<td>Document type</td>
<td>Selection</td>
</tr>
<tr>
<td>Overview of adding annotations</td>
<td>Free text</td>
</tr>
<tr>
<td>Annotation types recognised</td>
<td>Selection</td>
</tr>
<tr>
<td>Overview of adapting annotations</td>
<td>Free text</td>
</tr>
<tr>
<td>Change type supported</td>
<td>Selection</td>
</tr>
<tr>
<td>Usability study results</td>
<td>Free text</td>
</tr>
<tr>
<td>Integrated Development Environment integration</td>
<td>Free text</td>
</tr>
</tbody>
</table>

2.1.5 Data Extraction (Step 6)

The final list of publications was analysed using the data extraction form (see Table 2.1) to collect the details on each publication. This form has two main sections: general data and system data. The items in the general data section are based on the systematic mapping protocol (Okoli and Schabram, 2010). The items in the system data section, described below, are based on the research questions.

Overview

This is a summary of the publication. It includes what the implementation was attempting to do, what was actually achieved and what was involved.

Input mechanism

The Input mechanism is how the ink was physically collected. This started off as two options: tablet with stylus and Anoto pen on paper. As additional input mechanisms were found the list was expanded.
Application domain

The *Application domain* is the target domain of the application. This is a free text field to allow for any options. During data synthesis, this list was consolidated (see §2.1.6).

Document type

The *document type* describes the format of the text-based document. Based on the initial protocol development the starting values for this were ’text only’, ’Word’ and ’PDF’. As additional formats were found they were added to the list of values.

Overview of adding annotations

*Adding annotations* is the process of collecting digital ink strokes and associating them with the document. This overview lists the reported details on how an implementation handled adding annotations.

Annotation types recognised

An *annotation type* is a class of annotations that shares similar properties. For example, underlines are lines drawn underneath the text. Marshall (1997) proposed a number of annotation types and her types were used as the initial values. If a new type was found during the extraction phase it was added to the list of types. During data synthesis, this list was consolidated (see §2.1.6).

Overview of adapting annotations

*Adapting annotations* is the process of automatically modifying an annotation in response to a change in the document. This overview lists the reported details on any adaptations performed by the implementation.
Change type supported

Change type supported defines how the underlying document could change. This selection is based on the spectrum proposed by Golovchinsky and Denoue (2002): none; layout-only; and layout-and-content.

None means the implementation does not handle any changes to the document. This assumes that the document remains static throughout the lifetime of the annotations. The PDF format is an example of a format that does not allow changes. Layout-only means the rendering of the document can change but not the content. Examples of layout changes include changing the font size, page margin, zoom factor, etc. The ePub format is a format that allows for layout changes. Layout-and-content means that both the layout and the content of the document can change. For example, text can be added, modified or deleted, and images and other objects can be inserted or removed. Plain text is an example format for this change type. For some implementations, it was not possible to determine what types of changes are supported. These implementations were recorded as unknown.

Usability study results

These are the details of any human studies reported in the publication.

Integrated Development Environment integration

These are the details of any Integrated Development Environment integration. This could include how the implementation was implemented in the Integrated Development Environment, additional functionality and reported limitations (if any).

2.1.6 Data Synthesis (Step 7)

After data extraction, the publications were grouped by their implementation. Details were merged together when there were multiple publications about a single implementation. For the annotation types recognised field the values from all publications were included. The input mechanism and change type supported fields

\footnote{There are now tools that allow changing a PDF document but the original intention of the specification was for read-only documents.}
used the values from the most recent publication. For the remaining fields, the publications were compared. If there were details mentioned in one publication but not another they were combined. If there were conflicting details the details from the most recent publication were used.

The list of *annotation types recognised* was consolidated. Some publications used different terms for the same type of annotation (e.g. circling, enclosure and box are all synonymous). Each term was checked to see if they referred to the same *annotation type*. If so they were combined and the most common term used.

Next, the *addition* and *adaptation process* overviews were analysed. From each overview, all the operations that were mentioned were listed. For example, for addition a publication might mention “combining strokes” and “linking to underlying context”. This produced one list of operations for adding an annotation to the document and a second list of operations for adapting annotations.

The lists were then reviewed to find operations that were similar. For example, “combining strokes” and “grouping strokes” both refer to the same operation. Grouping similar operations together produced one set of operations for adding annotations and another for adapting them. To check the completeness of each set, all the implementations were reviewed to determine which operations were implemented.

During this process, the mode of anchoring an annotation to the underlying document was discovered as being important for adding annotations so the data synthesis step was repeated to capture the different anchor modes.

Finally, the results from *input mechanisms* and *application domains* were consolidated. Each item was summarised in a few words that described the main feature (e.g. “Presenting and annotating slides” was summarised as “Lecture Presentation”). These summaries were then grouped together where possible. If the summaries referred to the same feature, then they were combined (e.g. “Writing documents” and “Editing documents” were combined). Finally, the summaries were grouped into relevant hierarchies.
2.2 Results

2.2.1 Search Results

A total of 801 publications were found during the initial search phase. The selection process described in step 5 of the methodology was then applied (see §2.1.4). Out of the 801 publications, 48 met the inclusion criteria for the study. These publications were used as input to the forward and backward search and found an additional 578 publications. These publications were also checked against the exclusion and inclusion criteria and identified a further 13 publications to include. The final search (19 May 2016) found 16 additional publications. Of these publications, only two met the criteria to be included.

A total of 63 publications were included in the mapping study. These publications describe 43 different implementations. Most implementations are described by a single publication. Seven implementations have two publications (CodeAnnotator, CodeGraffiti, CoScribe, OneNote, Papiercraft, PenMarked and United slates), three implementations have three publications (Classroom Presenter, RCA and WriteOn) and XLibris has ten publications.

<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Implementation</th>
<th>Input Mechanism</th>
<th>Application Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>1991</td>
<td>Levine and Ehrlich</td>
<td>FreeStyle</td>
<td>Digitizer</td>
<td>Collaboration</td>
</tr>
<tr>
<td>1993</td>
<td>Hardock, Kurtenbach, and Buxton</td>
<td>MATE</td>
<td>Digitizer</td>
<td>Editing documents</td>
</tr>
<tr>
<td>1998</td>
<td>Price, Golovchinsky, and Schilit</td>
<td>Xlibris</td>
<td>Tablet PC</td>
<td>Active reading</td>
</tr>
<tr>
<td>1998</td>
<td>Schilit, Golovchinsky, and Price</td>
<td>Xlibris</td>
<td>Tablet PC</td>
<td>Active reading</td>
</tr>
<tr>
<td>1998</td>
<td>Schilit, Price, and Golovchinsky</td>
<td>Xlibris</td>
<td>Tablet PC</td>
<td>Active reading</td>
</tr>
<tr>
<td>1999</td>
<td>Golovchinsky, Price, and Schilit</td>
<td>Xlibris</td>
<td>Tablet PC</td>
<td>Active reading</td>
</tr>
<tr>
<td>1999</td>
<td>Marshall, Price, Golovchinsky, and Schilit</td>
<td>Xlibris</td>
<td>Tablet PC</td>
<td>Active reading</td>
</tr>
<tr>
<td>1999</td>
<td>Truong, Abowd, and Brotherton</td>
<td>Classroom 2000</td>
<td>Tablet PC</td>
<td>Lecture presentation</td>
</tr>
<tr>
<td>2000</td>
<td>Golovchinsky and Marshall</td>
<td>Xlibris</td>
<td>Tablet PC</td>
<td>Active reading</td>
</tr>
<tr>
<td>2000</td>
<td>Golovchinsky and Marshall</td>
<td>Xlibris</td>
<td>Tablet PC</td>
<td>Active reading</td>
</tr>
<tr>
<td>2001</td>
<td>Marshall, Price, Golovchinsky, and Schilit</td>
<td>Xlibris</td>
<td>Tablet PC</td>
<td>Active reading</td>
</tr>
<tr>
<td>2002</td>
<td>Golovchinsky and Denoue</td>
<td>Xlibris</td>
<td>Tablet PC</td>
<td>Active reading</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Implementation</td>
<td>Input Mechanism</td>
<td>Application Domain</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>----------------</td>
<td>----------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>2002</td>
<td>Götze, Schlechtweg, and Strothotte</td>
<td>Intelligent pen</td>
<td>Unknown</td>
<td>Active reading</td>
</tr>
<tr>
<td>2002</td>
<td>Mackay, Pothier, Letondal, Bøegh, and Sørensen</td>
<td>A-book</td>
<td>Multiple</td>
<td>Biology lab</td>
</tr>
<tr>
<td>2003</td>
<td>Bargeron and Moscovich</td>
<td>Callisto</td>
<td>Tablet PC</td>
<td>Not specified</td>
</tr>
<tr>
<td>2003</td>
<td>Guimbretière</td>
<td>PADD</td>
<td>Anoto</td>
<td>Not specified</td>
</tr>
<tr>
<td>2003</td>
<td>Shipman, Price, Marshall, and Golovchinsky</td>
<td>Xlibris</td>
<td>Tablet PC</td>
<td>Active reading</td>
</tr>
<tr>
<td>2004</td>
<td>Anderson, Anderson, Simon, Wolfman, VanDeGrift, and Yasuhara</td>
<td>Classroom Presenter</td>
<td>Tablet PC</td>
<td>Lecture presentation</td>
</tr>
<tr>
<td>2004</td>
<td>Anderson, Hoyer, Prince, Su, Videoen, and Wolfman</td>
<td>Classroom Presenter</td>
<td>Tablet PC</td>
<td>Lecture presentation</td>
</tr>
<tr>
<td>2004</td>
<td>Conroy, Levin, and Guimbretière</td>
<td>ProofRite</td>
<td>Multiple</td>
<td>Editing documents</td>
</tr>
<tr>
<td>2004</td>
<td>Shilman and Wei</td>
<td>DIZI</td>
<td>Tablet PC</td>
<td>Not specified</td>
</tr>
<tr>
<td>2005</td>
<td>Agrawala and Shilman</td>
<td>Classroom Presenter</td>
<td>Tablet PC</td>
<td>Lecture presentation</td>
</tr>
<tr>
<td>2005</td>
<td>Anderson, McDowell, and Simon</td>
<td>Classroom Presenter</td>
<td>Tablet PC</td>
<td>Lecture presentation</td>
</tr>
<tr>
<td>2005</td>
<td>Dontcheva, Drucker, and Cohen</td>
<td>v4v</td>
<td>Tablet PC</td>
<td>Lecture presentation</td>
</tr>
<tr>
<td>2005</td>
<td>Kam, Wang, Iles, Tse, Chiu, Glaser, Tarshish, and Canny</td>
<td>LiveNotes</td>
<td>Tablet PC</td>
<td>Lecture presentation</td>
</tr>
<tr>
<td>2005</td>
<td>Liao, Guimbretière, and Hinckley</td>
<td>Papercraft</td>
<td>Anoto</td>
<td>Active reading</td>
</tr>
<tr>
<td>2006</td>
<td>Chatti, Sodhi, Specht, Klamma, and Klemke</td>
<td>u-Annotate</td>
<td>Unknown</td>
<td>Web browsing</td>
</tr>
<tr>
<td>2006</td>
<td>Plimmer and Mason</td>
<td>PenMarked</td>
<td>Tablet PC</td>
<td>Marking</td>
</tr>
<tr>
<td>2006</td>
<td>Plimmer, Grundy, Hosking, and Priest</td>
<td>RCA</td>
<td>Tablet PC</td>
<td>Program code</td>
</tr>
<tr>
<td>2006</td>
<td>Priest and Plimmer</td>
<td>RCA</td>
<td>Tablet PC</td>
<td>Program code</td>
</tr>
<tr>
<td>2006</td>
<td>Tront, Eligeti, and Prey</td>
<td>WriteOn</td>
<td>Tablet PC</td>
<td>Lecture presentation</td>
</tr>
<tr>
<td>2006</td>
<td>Tront, Eligeti, and Prey</td>
<td>WriteOn</td>
<td>Tablet PC</td>
<td>Lecture presentation</td>
</tr>
<tr>
<td>2006</td>
<td>Wang, Shilman, and Raghupathy</td>
<td>OneNote</td>
<td>Tablet PC</td>
<td>Not specified</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Implementation</td>
<td>Input Mechanism</td>
<td>Application Domain</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>----------------</td>
<td>----------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>2007</td>
<td>Chen and Plimmer</td>
<td>CodeAnnotator</td>
<td>Tablet PC</td>
<td>Program code</td>
</tr>
<tr>
<td>2007</td>
<td>Liao, Guimbretière, Anderson, Linnell, Prince, and Razmov</td>
<td>PaperCP</td>
<td>Multiple</td>
<td>Lecture presentation</td>
</tr>
<tr>
<td>2007</td>
<td>Plimmer and Apperley</td>
<td>PenMarked</td>
<td>Tablet PC</td>
<td>Marking</td>
</tr>
<tr>
<td>2007</td>
<td>Signer and Norrie</td>
<td>PaperPoint</td>
<td>Anoto</td>
<td>Lecture presentation</td>
</tr>
<tr>
<td>2007</td>
<td>Wang and Raghupathy</td>
<td>OneNote</td>
<td>Tablet PC</td>
<td>Lecture presentation</td>
</tr>
<tr>
<td>2007</td>
<td>Cattelan, Teixeira, Ribas, Munson, and Pimentel</td>
<td>Inkteractors</td>
<td>Tablet PC</td>
<td>Not specified</td>
</tr>
<tr>
<td>2008</td>
<td>Chang, Chen, Priest, and Plimmer</td>
<td>RCA &amp; CodeAnnotator</td>
<td>Tablet PC</td>
<td>Program code</td>
</tr>
<tr>
<td>2008</td>
<td>Liao, Guimbretière, Hinckley, and Hollan</td>
<td>Papiercraft</td>
<td>Anoto</td>
<td>Active reading</td>
</tr>
<tr>
<td>2008</td>
<td>Weibel, Ispas, Signer, and Norrie</td>
<td>PaperProof</td>
<td>Anoto</td>
<td>Editing documents</td>
</tr>
<tr>
<td>2009</td>
<td>Chandrasekar, Tront, and Prey</td>
<td>WriteOn</td>
<td>Tablet PC</td>
<td>Lecture presentation</td>
</tr>
<tr>
<td>2009</td>
<td>Steimle</td>
<td>CoScribe</td>
<td>Anoto</td>
<td>Studying</td>
</tr>
<tr>
<td>2010</td>
<td>Lichtschlag and Borchers</td>
<td>CodeGraffiti</td>
<td>Capacitive touch</td>
<td>Program code</td>
</tr>
<tr>
<td>2010</td>
<td>Plimmer, Chang, Doshi, Laycock, and Seneviratne</td>
<td>iAnnotate</td>
<td>Tablet PC</td>
<td>Web browsing</td>
</tr>
<tr>
<td>2012</td>
<td>Chen, Guimbretière, and Sellen</td>
<td>United slates</td>
<td>elink Reader</td>
<td>Active reading</td>
</tr>
<tr>
<td>2012</td>
<td>Hinckley, Bi, Pahud, and Buxton</td>
<td>GatherReader</td>
<td>Tablet PC</td>
<td>Active reading</td>
</tr>
<tr>
<td>2012</td>
<td>Matulic and Norrie</td>
<td>Matulic &amp; Norrie (2012)</td>
<td>Hybrid</td>
<td>Active reading</td>
</tr>
<tr>
<td>2012</td>
<td>Steimle</td>
<td>CoScribe</td>
<td>Anoto</td>
<td>Collaboration</td>
</tr>
<tr>
<td>2013</td>
<td>Bhardwaj, Chaudhury, and Roy</td>
<td>Augmented Paper System</td>
<td>Visual</td>
<td>Not specified</td>
</tr>
<tr>
<td>2013</td>
<td>Chen, Guimbretière, and Sellen</td>
<td>United slates</td>
<td>elink Reader</td>
<td>Active reading</td>
</tr>
<tr>
<td>2013</td>
<td>Marinai</td>
<td>Marinai (2013)</td>
<td>Unknown</td>
<td>Not specified</td>
</tr>
<tr>
<td>2013</td>
<td>Mazzei, Blom, Gomez, and Dillenbourg</td>
<td>annOot</td>
<td>Tablet PC</td>
<td>Studying</td>
</tr>
</tbody>
</table>
Table 2.2 lists the publications that were included in the mapping study, together with their high-level details. Full bibliographic information for all publications is in the reference list.

### 2.2.2 General Information

The earliest publication identified was published in 1991. Since then there have been between zero and five publications published each year (see Figure 2.1).

This section covers the input mechanisms, change types supported, application domains and document formats.

**Input Mechanism**

The input mechanism was identified for 38 publications (see Figure 2.2). The following input mechanisms were found: Tablet PC with stylus; Anoto pen; digitizer with stylus; PDA with stylus; customised eInk readers; capacitive touch and visual input.

The most common input mechanism reported is a stylus on a tablet PC (20 implementations). In this mechanism, the user directly draws on the screen of the tablet PC with a special stylus. The tablet PC directly captures the digital ink which is then processed by the implementation. Eight implementations used an Anoto pen on paper. The document is printed and an Anoto pen records inking on the paper. The ink is then converted to a digital form, loaded into the implementation and added to the original digital document. A further two
implementations allowed input from either the Anoto pen or a stylus on a tablet PC. The remaining eight implementations used a variety of input mechanisms. Four used a stylus: two via a digitizer (Levine and Ehrlich, 1991, Hardock et al., 1993), one with a PDA (Mackay et al., 2002) and one used customised eInk readers (Chen et al., 2012, 2013). Two used touch on a capacitive touch surface (Lichtschlag and Borchers, 2010, Pearson and Buchanan, 2010). One used an Anoto Pen combined with a tabletop PC (Matulic and Norrie, 2012) and the final implementation used visual input to track the tip of a pen (Bhardwaj et al., 2013).

There were three dimensions influenced by the input mechanism:

(i) directness
(ii) accuracy
(iii) physical size

Directness describes the relationship between the input surface and the display. A direct mechanism (Tablet PCs and Anoto Pens) involves direct interaction with the display surface: the surface the user is interacting with is the surface where the content is displayed\(^3\). In contrast, with an indirect mechanism (digitizer) there is a disconnect between the input surface and the document. The user has

\(^3\)Anoto Pens are both a direct and an indirect mechanism (mixed directness). When the user is annotating they are directly annotating on the display. When the annotations are displayed on a computer monitor the mechanism is indirect.
to map from the document to the input surface. With directness, there are two possible interactions: input and output. Tablet PCs provide direct interaction for both input and output. The user can directly input ink onto the device and see it update. In contrast, Anoto Pens provide mixed directness: while the user can directly input on the display surface and see it update on the display surface, they do not directly see its digital representation. Digitizers also provide mixed directness: for input they are indirect but the user can directly see the digital representation. No mechanisms were identified that were indirect for both input and output.

Accuracy describes the level of precision when using the input mechanism. The most accurate input mechanism mentioned was the Anoto Pens. These have a theoretical precision of 0.03mm (Matulic and Norrie, 2012). The least accurate input mechanism is using touch on a capacitive surface. Stylus input devices have a range of accuracies but few publications record any details on the level of precision achieved. However, the precision will be lower than an Anoto due to the decreased display resolution compared to paper (Agrawala and Shilman, 2005).

Physical size refers to the physical size of the input surface. The smallest device was the PDA for A-book (Mackay et al., 2002). The next larger devices are Tablet PC and eInk reader systems. Finally, the largest physical systems are the tabletop implementations. These systems require space for the table plus additional ancillary equipment (e.g. the camera in Bhardwaj et al., 2013). Anoto pen systems can potentially be anywhere on this dimension as the input surface depends on the size of the paper the document is printed on.
Change Type Supported

Of the 43 implementations, 3 supported layout-only changes, 11 supported layout-and-content changes and 36 did not support any type of change in the underlying document. For the remaining 12, it was not possible to determine whether they supported any changes. The definitions of each change type are defined in §2.1.5.

Application Domains

During the analysis five main domains were identified:

(i) Collaboration
(ii) General
   (1) Active reading
   (2) Document editing
   (3) Web browsing
(iii) Education
   (1) Lecturing
   (2) Marking
   (3) Studying
(iv) Programming
(v) Research

Table 2.2 includes the application domain for each implementation.

Collaboration systems are primarily intended for communications between two or more people. In these systems, annotations are a way of communicating information. For example, Wang Freestyle allowed people to exchange notes and documents. Annotations enhanced document exchange by including a simple way to add additional information (Levine and Ehrlich, 1991).

General covers both reading and producing documents. The most common category in this document is reading documents: specifically active reading⁴. During active reading, the reader uses a pen to mark the text as they read. XLibris, the implementation with the most publications, was originally designed as an active reading device (Schilit et al., 1998a). Web browsing is another form of reading but with some key differences: active reading applications replicate

⁴As mentioned by Chen et al. (2012) active reading also implies similar terms such as work-related reading and responsive reading.
how paper works (Price et al., 1998b, Schilit et al., 1998a,b) while web browsing focuses on the dynamic nature of web pages (Chatti et al., 2006, Plimmer et al., 2010). Finally, document editing refers to the process of producing and editing documents. The three implementations in this category all focused on how an editor can annotate a document (Hardock et al., 1993, Conroy et al., 2004, Weibel et al., 2008).

The most common area in Education is lecturing: presenting slides to a class with annotation support (Truong et al., 1999, Anderson et al., 2004a, 2005, Kam et al., 2005). Some implementations investigated how annotations can help students when studying and taking notes (Steimle et al., 2009, Mazzei et al., 2013). One implementation investigated how annotations improved marking of student work (Plimmer and Mason, 2006, Plimmer and Apperley, 2007).

The final two domains are subject specific and have a limited number of implementations. Some implementations looked at how annotations could be added to program code. These implementations mainly focused on the technical challenges of adding annotations within current Integrated Development Environments and how they can assist with navigation (Plimmer et al., 2006, Priest and Plimmer, 2006, Chen and Plimmer, 2007). However, one also looked at how annotations could be useful for code in a collaborative environment (Lichtschlag and Borchers, 2010). The other domain was research. The single implementation in this domain looked at how annotations provide a link between physical and electronic documents (Mackay et al., 2002).

Document Formats

Many implementations do not mention document formats used (17 out of 43). The formats that are mentioned can be grouped into nine formats (see Figure 2.3). The most common formats mentioned were HTML (Bargeron and Moscovich, 2003, Ramachandran and Kashi, 2003, Chatti et al., 2006, Wu et al., 2008, Steimle, 2009, Steimle et al., 2009, Plimmer et al., 2010) and PDF (Guimbretière, 2003, Steimle, 2009, Steimle et al., 2009, Chen et al., 2012, 2013, Marinai, 2013, Yoon et al., 2013, Asai and Yamana, 2014) (seven implementations each) and then program code (Plimmer et al., 2006, Priest and Plimmer, 2006, Chen and Plimmer, 2007, Chang et al., 2008, Lichtschlag and Borchers, 2010, Sutherland and Plimmer, 2013, Lichtschlag et al., 2014), PowerPoint (Anderson et al., 2004a, Signer and Norrie, 2007, Steimle, 2009, Steimle et al., 2009) and scanned documents (Levine
and Ehrlich, 1991, Price et al., 1998a, Schilit et al., 1998a,b, Golovchinsky et al., 1999, Marshall et al., 1999, Golovchinsky and Marshall, 2000a,b, Marshall et al., 2001, Golovchinsky and Denoue, 2002, Mackay et al., 2002, Shipman et al., 2003, Steimle, 2009) (all with four implementations). Scanned documents refers to documents that have been scanned and loaded into the implementation. Rich Documents refers to documents produced by word processing software: two implementations use this format (AbiWord (Conroy et al., 2004) and OpenOffice (Weibel et al., 2008)). Screen capture refers to a direct capture of the screen: two implementations use this format (Olsen et al., 2004, Tront et al., 2006a,b, Chandrasekar et al., 2009).

2.2.3 Integrated Development Environment Integration

Of special relevance to this thesis is how annotations have been added to Integrated Development Environments and for what purpose. This shows what has been investigated before, what has worked and what has not, and the details on how.

My search identified four implementations that added annotations to Integrated Development Environments. These implementations are:
(i) RCA – *Visual Studio 2005* (Plimmer et al., 2006, Priest and Plimmer, 2006, Chang et al., 2008);
(ii) CodeAnnotator – *Eclipse* (Chen and Plimmer, 2007, Chang et al., 2008);
(iii) CodeGraffiti – *XCode* and *Adobe Brackets* (Lichtschlag and Borchers, 2010, Lichtschlag et al., 2014);
(iv) vsInk – *Visual Studio 2010* (Sutherland and Plimmer, 2013).

**Technical Details**

The focus of publications on RCA, CodeAnnotator and vsInk was how to implement freeform ink annotations in an Integrated Development Environment. Neither RCA nor CodeAnnotator was able to directly integrate freeform ink annotations into the code editor (Chang et al., 2008). Instead, they both used a dual window approach. One window contained the code editor: users had full access to the editor functionality in this window. The second window was the inking window. It contained a read-only copy of the code that the user could annotate on. The two windows were linked but the user had to manually trigger a transfer of code from the editor to the inking window (Priest and Plimmer, 2006). Thus, while the user could annotate on the code it was a copy rather than the actual code.

The main reason for this limitation was the limited extension points with the two Integrated Development Environments investigated (*Visual Studio* and *Eclipse*) (Chang et al., 2008). With the release of *Visual Studio 2010*, additional extension points were added in the Integrated Development Environment that allowed direct integration with the code editor. vsInk demonstrated how these extension points could be used to add freeform digital ink directly to the code editor itself (Sutherland and Plimmer, 2013).

The publications on these three implementations provide details on how freeform ink annotations are added to the Integrated Development Environments (Priest and Plimmer, 2006, Chen and Plimmer, 2007, Sutherland and Plimmer, 2013). All three implementations anchor the annotations to a line of code. The type of the initial stroke, the linker, is used to determine how the annotation is anchored. RCA and CodeAnnotator both identify two types of stroke (line and circles) using heuristic rules (Priest and Plimmer, 2006, Chen and Plimmer, 2007). vsInk

---

5 An extension point is a defined location in the Integrated Development Environment that can be extended by code from an external author.
recognised six linker types using a machine learning approach, RATA, as the classification mechanism (Sutherland and Plimmer, 2013).

All three implementations provide some form of grouping strokes into an annotation. *RCA* uses three rules for grouping strokes. First, if subsequent strokes are added within two seconds of the previous stroke, they are grouped together. Second, if the stroke is added within the bounding box of the annotation, it is grouped with the existing strokes. Finally, the user can manually select a linker and force subsequent strokes to be grouped together (Priest and Plimmer, 2006). *CodeAnnotator* builds on these three rules and explored how the bounding box influences grouping of new strokes. The authors concluded additional space was required at the bottom and right sides of the annotation to allow grouping (Chen and Plimmer, 2007). *vsInk* also used the same temporal and spatial rules but expanded them to handle multiple annotations. The annotation with the closest centroid to the stroke's starting point was used if there were multiple candidates for grouping (Sutherland and Plimmer, 2013).

Finally, all three implementations used some form of adaptation (called reflow for all three implementations). All three would reposition the annotation based on the associated anchor. They specifically mention moving the annotation in response to adding or deleting lines above the annotation (Priest and Plimmer, 2006, Chen and Plimmer, 2007, Sutherland and Plimmer, 2013). In addition, all three implementations delete the annotation if the anchor line is deleted.

Publications on the fourth implementation, *CodeGraffiti*, do not provide any details on how freeform ink is implemented into the Integrated Development Environment (Lichtschlag and Borchers, 2010, Lichtschlag et al., 2014). Instead, the focus is on the functionality provided. *CodeGraffiti* went through two iterations: in each iteration, the extension was developed for a different Integrated Development Environment. The first iteration targeted *XCode* and allowed two participants to view and annotate the code at the same time. The second iteration targeted *Adobe Brackets* and added a “Mission Control” view where the user could draw diagrams and text and connect them to locations in the code.

**Functionality Implemented**

*RCA* primarily investigated whether it was possible to integrate digital ink in an Integrated Development Environment. This implementation proved that it was possible, even though it had limitations (Priest and Plimmer, 2006).
Table 2.3: Initial Taxonomy of Annotation Types.

<table>
<thead>
<tr>
<th>Type Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single line</td>
<td>Underlines</td>
</tr>
<tr>
<td></td>
<td>Highlighting</td>
</tr>
<tr>
<td>Multiple line</td>
<td>Enclosures</td>
</tr>
<tr>
<td></td>
<td>Margin bars</td>
</tr>
<tr>
<td></td>
<td>Braces</td>
</tr>
<tr>
<td>Connectors</td>
<td>Callouts/arrows</td>
</tr>
<tr>
<td>Complex</td>
<td>Text/symbols within text</td>
</tr>
<tr>
<td></td>
<td>Drawings</td>
</tr>
<tr>
<td></td>
<td>Marginalia</td>
</tr>
<tr>
<td>Commands</td>
<td>Commands</td>
</tr>
</tbody>
</table>

*CodeAnnotator* extended the results from *RCA* and added navigation support. This implementation suggested that navigation would be an important feature of inking in an Integrated Development Environment, although this was not tested (Chen and Plimmer, 2007). *CodeAnnotator* provided two forms of navigation: within a file and across files. The across file navigation allowed users to quickly see which files had been annotated and then the within file navigation allowed them to see the specific location of the annotation. The within file navigation also showed a graphical representation of the annotation.

*vsInk* also implemented navigation, although it only included within file navigation (Sutherland and Plimmer, 2013). The navigation again showed a graphical representation of the annotation and selecting the annotation took the user to its location. In addition, *vsInk* included support for collapsed regions. This was based on the hypothesis that should the annotation’s anchor be hidden then the entire annotation should also be hidden (Sutherland and Plimmer, 2013). While *vsInk* was evaluated with a user study it is unknown how effective the navigation is as this was not investigated.

Finally, *CodeGraffiti* investigated two different areas. The first iteration of *CodeGraffiti* investigated how ink could be used in a peer-programming environment. In the publication on this iteration, they described their iteration and how it allowed for a second programmer to view and annotate the code on a remote device. One issue they identified was how to handle concurrent navigation and annotation (Lichtschlag and Borchers, 2010).

The second iteration of *CodeGraffiti* focused on navigation. In this iteration the user had a blank document, mission control, where they could draw and
### Table 2.4: Taxonomy of Annotation Support Operations.

<table>
<thead>
<tr>
<th>Category</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding operations</td>
<td>Grouping</td>
</tr>
<tr>
<td></td>
<td>Recognising</td>
</tr>
<tr>
<td></td>
<td>Anchoring</td>
</tr>
<tr>
<td></td>
<td>Storing</td>
</tr>
<tr>
<td>Adapting operations</td>
<td>Repositioning</td>
</tr>
<tr>
<td></td>
<td>Refitting</td>
</tr>
<tr>
<td></td>
<td>Orphaning/Deleting</td>
</tr>
</tbody>
</table>

diagram. This diagram could then be linked to the source code. These links were bi-directional and allowed the user to view both the source location and the associated diagram position. In their study, they found the participants with *CodeGraffiti* spent more time looking at the diagrams. However, this did not appear to influence accuracy or task completion time (Lichtschlag et al., 2014).

#### 2.2.4 Taxonomy

A summary of the research approaches to digital ink software is presented in Tables 2.3 and 2.4. These taxonomies have been compiled as a result of the data synthesis step (see §2.1.6). Table 2.3 lists the categories of annotation types and examples of each type. Table 2.4 lists the adding and adapting operations.

#### Annotation Types Recognised

It is clear from the various studies that have been conducted (e.g. Marshall, 1997, Golovchinsky and Denoue, 2002, Marshall and Brush, 2002, Bargeron and Moscovich, 2003, Shilman and Wei, 2004, Wang et al., 2006, Sutherland et al., 2015) that there are some common types of annotations. However, these are dependent on both the individual and the domain. Despite this limitation, many implementations still attempt to recognise the annotation type as this allows for additional functionality later.

The annotation types recognised could be determined for 24 implementations. The remainder of the implementations either do not handle specific annotation types.

---

6The initial proposal was based on previously reported studies. See §6.1.1 in Chapter 6 for the revised taxonomy.
### Table 2.5: Annotation category recognised by implementation.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Single Line</th>
<th>Multiple Line</th>
<th>Complex</th>
<th>Connector</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeGraffiti (Lichtschlag et al., 2014), Callisto (Bargeron and Moscovich, 2003)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Classroom Presenter (Anderson et al., 2004a,b, 2005)</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeAnnotator (Chen and Plimmer, 2007, Chang et al., 2008)</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intelligent ink annotation framework (Asai and Yamana, 2014)</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intelligent pen (Götze et al., 2002)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MATE (Hardock et al., 1993)</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Matulic &amp; Norrie (2012) (Matulic and Norrie, 2012)</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>OneNote (Wang et al., 2006, Wang and Raghupathy, 2007)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>PaperCP (Liao et al., 2007)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>PaperPoint (Signer and Norrie, 2007)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>PaperProof (Weibel et al., 2008)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Papiercraft (Liao et al., 2005, 2008)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>ProofRite (Conroy et al., 2004)</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCA (Plimmer et al., 2006, Priest and Plimmer, 2006, Chang et al., 2008)</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>ScreenCrayons (Olsen et al., 2004)</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shilman &amp; Wei (2004) (Shilman and Wei, 2004)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Steimle (2009) (Steimle, 2009)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>United slates (Chen et al., 2012, 2013)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>vslnk (Sutherland and Plimmer, 2013)</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Wu, Yang &amp; Su (2008) (Wu et al., 2008)</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Yoon, Chen &amp; Guimbretière (2013) (Yoon et al., 2013)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
</tbody>
</table>

Number of Implementations: 10 12 8 7 12
types or do not describe the annotation types recognised. Table 2.5 lists these implementations and the categories that they recognise.

Ten different annotation types were identified. The following are the definitions of each annotation type:

(i) An underline is a line drawn underneath or through a sentence.
(ii) A highlight is similar to an underline but drawn with a different (often semi-transparent) pen.
(iii) An enclosure is a border around one or more elements.
(iv) A margin bar is a vertical straight line drawn in a margin.
(v) A brace is similar to a margin bar but has a pronounced rounded shape and a centre prominence.
(vi) An arrow or call out is a line drawn from one element to another. It may have arrowheads on one or both end points.
(vii) Text and symbols are characters written in the body of the underlying text. They are generally added in the whitespace around the underlying text.
(viii) A drawing is a picture or diagram.
(ix) Marginalia are longer notes added in the margin.
(x) Commands are marks that the implementation is expected to understand and execute.

Figure 2.4 shows the breakdown of which annotation types are commonly supported.

Different annotation types have different requirements from a software support perspective. These annotation types were grouped into five categories based on the requirements of each type:

(i) Single line (underline and highlighting): these annotations are associated with a single line in the document.
(ii) Multiple line (enclosures, margin bars and braces): these annotations span multiple lines.
(iii) Connectors (arrows/callouts): these annotations associate two areas or annotations together.
(iv) Complex (text/symbols, drawings and marginalia): these annotations have an additional meaning in addition to their location.
(v) Commands (commands): these marks are commands for the system to perform. These are usually a limited set of symbols that the system can recognise.
Chapter 2. Literature Review

**Figure 2.4:** Number of implementations that recognise each category of annotations.
Table 2.6 shows the five categories and an example of each of the types.

The main difference between the first two categories is the level of association. Single line annotations are associated with the words in a continuous line of text. In contrast, multiple line annotations are more general and are associated with the entire line, rather than the individual words⁷.

<table>
<thead>
<tr>
<th>Single Line Annotations</th>
<th>Multiple Line Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Underline</td>
<td>Highlight</td>
</tr>
<tr>
<td>HTML DOM elements</td>
<td></td>
</tr>
<tr>
<td>element (if available), other consistent robust anchor;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Connector Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callout/arrow</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Complex Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

---

⁷Chapter 5 discusses a scenario where these categories fail; however, the categories in this chapter describe how these annotations have been implemented previously.
Table 2.6: Continued.

<table>
<thead>
<tr>
<th>Commands</th>
<th>Drawing</th>
<th>Marginalia</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Year of publication;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Venue;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Authors;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Title;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Source</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Single line annotations were recognised by ten implementations. All ten implementations recognised underlines; six also recognised highlights.

Multiple line annotations were recognised by twelve implementations. All twelve implementations recognised enclosures, seven recognised margin bars and three recognised braces.

Connectors were recognised by seven implementations.

Complex annotations are recognised by six implementations. All six implementations especially mentioned they recognised text/symbols; two recognised drawings and three recognised marginalia. These three implementations mentioned marginalia as a separate type; other implementations may have supported the same type but reported them as text/symbols.

Finally, commands were recognised by twelve implementations.

In addition to these categories, annotations fit into two classes based on their intended use. The first class of annotations are those intended for a person. While a computer may recognise these annotations they often have additional meaning beyond their appearance and location. The second class of annotations are those intended for the computer. These annotations can be completely understood by the computer. All commands fall into this category as the computer must understand them to apply them.

One challenge with annotation systems is dividing annotations into the two classes. If the annotation is intended for the application, there needs to be some way of recognising these annotations; otherwise, the application will treat them
as intended for a human. The publications reviewed in this study describe the following approaches:

(i) Pen buttons
(ii) Separate display space
(iii) Special gestures
(iv) Pen plus touch

Pen buttons involve using one or more buttons on the stylus device. When the user wants to change modes they depress these buttons. The buttons can either change the mode until the button is pressed again or only change the mode while the button is depressed. With the second option, the mode returns to the original when the button is released.

With a separate display space, there is an area of the screen where gestures must be entered. Any ink outside this area is assumed to be human-readable only. Gestures within this area will be recognised and potentially processed.

Special gestures involve either a specific gesture set or a special gesture that is added to other gestures. With the specific gesture set, the implementation attempts to recognise all gestures. If the gesture is recognised then it will be processed; otherwise, the implementation assumes that it is human-readable ink instead. With the special gesture, the implementation will ignore all ink unless it includes the gesture. If the special gesture is included, then the entire gesture is assumed to be a command.

Finally, with pen plus touch, one hand is used to control the pen and another to provide touch input to the implementation. Based on the touch input, the implementation either treats the pen input as human readable or as commands.

**Adding Operations**

Adding an annotation to a document involves several steps. Digital ink is captured as ink strokes. A single ink stroke is generated by a pen-down, pen moves, pen-up sequence. However, people do not consider ink strokes individually. Instead, they mentally group them together as annotations. An annotation can consist of a single ink stroke (e.g. an underline or enclosure) or multiple strokes (e.g. text or drawings).

I found four operations involved in adding an annotation to a document:
(i) Grouping: combining multiple ink strokes into a single annotation;
(ii) Recognition: classifying all or part of the annotation;
(iii) Anchoring: determining the location of the annotation relative to the underlying context (further details are mentioned in §2.2.4);
(iv) Storage: persisting the annotation details, including information about the anchor.

Very few publications reported on how annotations are stored. Most implementations that store the annotations use a separate file for the annotations (i.e. they do not modify the original document). These files are either stored locally on the user’s machine (Chatti et al., 2006, Priest and Plimmer, 2006, Sutherland and Plimmer, 2013) or sent to a server (Plimmer et al., 2010). Another approach is to modify the original file to store the annotations in it (Chen et al., 2012, Wang et al., 2006, Wang and Raghupathy, 2007).

Binary and XML-based formats are the only two storage formats mentioned. Only one publication describes the storage format in detail (Ramachandran and Kashi, 2003) although some other implementations do use pre-defined formats (e.g. Microsoft’s ISF format) (Priest and Plimmer, 2006, Sutherland and Plimmer, 2013).

Different implementations use different sequences of these steps. The sequence of steps implemented depends on the goals of the implementation. However, there are two general sequences: all-annotation and single-annotation.

The all-annotation sequence processes all strokes in the document when a new stroke is added. The first step is to group strokes into annotations. Then the implementation attempts to recognise each annotation. The result from the recognition (the annotation type) is used to anchor the annotation in the document. Finally, the annotation details are stored. With this sequence, there is more information for the grouping and recognition steps. This may improve recognition accuracy but comes at the cost of increased computation. This increase is due to the need to reprocess all strokes. The implementations by Shilman and Wei (2004), Wang et al. (2006) and Wang and Raghupathy (2007) are examples of this sequence.

The single-annotation sequence processes each stroke only once. When a stroke is added, the first step is to group the stroke with an existing annotation. If this is not possible then a new annotation is started. The implementation recognises the annotation and anchors it to the document. Recognition and anchoring are
only applied to new annotations. No matter how many strokes are added to an existing annotation, the type and anchor do not change. Finally, the annotation is stored. This sequence requires less rework as recognition and anchoring are only performed once for the entire annotation but this may reduce recognition and anchoring precision.

The single-annotation sequence can involve the user in the grouping operation. The amount of user involvement ranges from the user manually doing all the grouping to the implementation providing hints. Callisto requires the user to do the grouping (and recognition) (Bargeron and Moscovich, 2003). RCA, Code-Annotator and vsInk all provide grouping hints by displaying a border around the annotations. When the new stroke is inside or intersects an annotation’s border it is included with the annotation (Chen and Plimmer, 2007, Priest and Plimmer, 2006, Sutherland and Plimmer, 2013). XLibris does not provide any user feedback or involvement in the grouping (Golovchinsky and Denoue, 2002). Instead, it uses timing and spatial heuristics to automatically group strokes together.

Not all implementations use all four operations. Some implementations do not mention any form of recognition (e.g. Chatti et al., 2006, Plimmer et al., 2010). Other implementations treat all strokes as individual annotations and do not mention any grouping (e.g. Shilman and Wei, 2004, Wu et al., 2008).

There are also notable exceptions to the overall sequences listed above. These typically include one or more of the steps but don’t do it for the purpose of adding annotations. For example, recognition is used to separate temporal, attention strokes from permanent ink annotations (Anderson et al., 2004b), to identify sections in a document that would be most useful to the user (Shipman et al., 2003) and to apply a mask to the document to emphasize what was annotated (Olsen et al., 2004).

There have not been any comparative studies between these sequences to determine the relative efficacy of each.

**Anchoring Mode**

Annotation anchoring involves associating an annotation with an element of context in the document. All current research treats freeform ink annotations as graphical elements. These graphical elements are positioned in the document
Table 2.7: Number of implementations for each anchoring approach.

<table>
<thead>
<tr>
<th>Anchoring Approach</th>
<th>Number of Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whole page</td>
<td>20</td>
</tr>
<tr>
<td>HTML Element</td>
<td>4</td>
</tr>
<tr>
<td>Code Line</td>
<td>4</td>
</tr>
<tr>
<td>Unknown</td>
<td>12</td>
</tr>
<tr>
<td>Word</td>
<td>2</td>
</tr>
<tr>
<td>Paragraph</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>43</strong></td>
</tr>
</tbody>
</table>

relative to a bounding box. The anchor mode is classified by the type of bounding box used (see Table 2.7).

The most common approach is to use the whole page as the bounding box. Both the document and the annotations are treated as graphical representations. Typically, the annotation’s top left corner is recorded as an offset from the top left of the page. The annotations are merged into the document to produce the final view. While this approach is simple and easy to implement, it does not allow for the underlying document to change. If the document changes, a new graphical representation needs to be generated and the associated coordinates either lose their meaning or need to be translated. There are no reports of translating an annotation to new coordinates without using a more sophisticated approach to anchoring.

The remaining approaches all use a smaller element on the page. The page is decomposed into these elements and the closest element is selected as the bounding box. Where there are multiple choices the implementation will use some form of preferential ordering to select the “best” bounding box (Wang and Raghupathy, 2007, Sutherland and Plimmer, 2013).

Both paragraph and word approaches use the words in the document as the anchor. The anchor can include using the words themselves, using a number to identify the word within the document (e.g. words 10 to 15) or the location of the words (e.g. words 1 to 5 in the second paragraph of the third page) (Golovchinsky and Denoue, 2002). All of these approaches assume the words do not change within the document. This approach does support reflow of the existing text, for example, if the font size changes so that words flow onto other lines (Golovchinsky and Denoue, 2002).
Anchoring with an HTML element uses the underlying HTML document object model (DOM). HTML uses a tree-like structure for generating a page. The browser renders this structure into a graphical representation that the user sees. Choosing an anchor position involves finding the closest element to the annotation. The information stored for the annotation includes the identifier of the element (if any), the path from the root to the element and the surrounding elements (Chatti et al., 2006, Plimmer et al., 2010).

The bounding box for a code line is around each individual line. The anchor for code annotations consists of the line number and file name (Chen and Plimmer, 2007, Priest and Plimmer, 2006, Sutherland and Plimmer, 2013).

### Adapting Operations

When the underlying document changes, an annotation may need to adapt in response. By adapting the annotation, it retains its meaning and value. The actual type of adaptation depends on how the underlying document is modified.

There is less work published on automatically adapting annotations. Only ten out of the 31 implementations mention any form of automatic adaptation, these being grouped into three categories:

(i) Repositioning: the annotation is moved to a new location;
(ii) Refitting: the appearance of the annotation is changed;
(iii) Orphaning: the underlying context for the annotation has been removed.

In addition to these categories, implementations can be classified by the type of document modification that is handled. Table 2.8 shows the relationship between these two categories.

Nine implementations handle repositioning annotations when the underlying content changes, four handle orphaning and four refit annotations. There was

<table>
<thead>
<tr>
<th></th>
<th>Repositioning</th>
<th>Refitting</th>
<th>Orphaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout-only</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Layout-and-content</td>
<td>6</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>9</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2.8: Number of implementations that automatically adapt annotations.
only one implementation that handled refitting but not repositioning (Ramachandran and Kashi, 2003). This is unusual as normally repositioning is easier to implement than refitting. It may be this implementation does handle repositioning but it was not mentioned in the publication. All systems that implement orphaning, also implement repositioning.

Four out of the six implementations that handle content changes are code editors. The others are ProofRite and Callisto (Conroy et al., 2004, Bargeron and Moscovich, 2003). All four implementations that handle orphaning are also code editors. They all use the same approach for handling orphaning by deleting the annotation. None of the publications on these implementations mention any way for the user to review the orphaned annotations (Chen and Plimmer, 2007, Lichtschlag and Borchers, 2010, Plimmer et al., 2006, Priest and Plimmer, 2006, Sutherland and Plimmer, 2013).

The effectiveness of annotation repositioning is related to anchoring. Repositioning requires an anchoring mode at a more granular level than whole-page. All nine implementations that support repositioning use a more granular mode. Four implementations used a line bounding box (Priest and Plimmer, 2006, Chen and Plimmer, 2007, Lichtschlag and Borchers, 2010, Sutherland and Plimmer, 2013); one used a bounding box based on HTML elements (Plimmer et al., 2010); one used paragraph level bounding boxes (Marinai, 2013); two used word level bounding boxes (Golovchinsky and Denoue, 2002, Bargeron and Moscovich, 2003); the final implementation did not mention how the annotations were anchored to the context (Conroy et al., 2004).

Repositioning calculates the new position of the annotation using the position of the anchor element plus an offset. The first step is to retrieve the current location of the reference point. The offset is then added to this reference point and the annotation positioned using the sum. All approaches are, in reality, using \((x, y)\) coordinates (as the annotation is a graphical element) translated relative to the anchor (Sutherland and Plimmer, 2013).

Only four implementations mentioned any refitting of annotations: XLibris (Golovchinsky and Denoue, 2002); Callisto (Bargeron and Moscovich, 2003); ProofRite (Conroy et al., 2004) and the system by Ramachandran and Kashi (Ramachandran and Kashi, 2003). All of these implementations use similar rules. Two systems only handle layout changes (XLibris and Ramachandran and Kashi). For XLibris, the rationale was to remove any confounding influence due to not
finding an anchor for an annotation. *Callisto* and *ProofRite* handle both content and layout changes.

In *XLibris*, single-line annotations (underlines and highlighting) remain attached to the words they are anchored to. If a line splits, then the annotation will also split; if two lines with similar annotations are joined, the annotations will also be joined. Multi-line annotations (enclosures and margin bars) are stretched or condensed so the top and bottom margins of the annotations stay in the same relative positions to the underlying context. Complex annotations are not refitted (Golovchinsky and Denoue, 2002).

*Callisto* treats enclosures as a single line annotation and associates them with the line in the same way *XLibris* handles underlines and highlights. Braces are handled instead of margin bars. *Callisto* also has a mode where the annotations are converted to “cleaned” annotations. Underlines and highlights are converted to straight lines that align with the underlying text; enclosures are converted to rectangles with rounded corners and aligned to the underlying text; braces are converted to simple Bezier curves. Once cleaned, the annotation then follows the same refit rules as the original annotations. The rationale for this is it is easier to automatically refit “cleaned” annotations (Bargeron and Moscovich, 2003).

*ProofRite* follows the same rules as *XLibris* (Bargeron and Moscovich, 2003). The system by Ramachandran and Kashi (2003) does not describe the rules for adapting annotations.

Table 2.5 shows which implementations recognise the different categories of annotations. Table 2.9 shows the implementations which implement adding and adapting operations.

**Table 2.9: Adding and Adapting operations by implementation.**

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Group</th>
<th>Recognise</th>
<th>Anchor</th>
<th>Store</th>
<th>Reposition</th>
<th>Refit</th>
<th>Orphan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callisto (Bargeron and Moscovich, 2003)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Classroom Presenter</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Anderson et al., 2004a,b, 2005)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeAnnotator (Chen and Plimmer, 2007,</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Chang et al., 2008)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeGraffiti (Lichtschlag and Borchers,</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>2010)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td>Group</td>
<td>Recognise</td>
<td>Anchor</td>
<td>Store</td>
<td>Reposition</td>
<td>Refit</td>
<td>Orphan</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>--------</td>
<td>-------</td>
<td>------------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>CoScribe (Steimle et al., 2009, Steimle, 2012)</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iAnnotate (Plimmer et al., 2010)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intelligent ink annotation framework (Asai and Yamana, 2014)</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intelligent pen (Götze et al., 2002)</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Marinai (2013) (Marinai, 2013)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>MATE (Hardock et al., 1993)</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matulic &amp; Norrie (2012) (Matulic and Norrie, 2012)</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OneNote (Wang et al., 2006, Wang and Raghupathy, 2007)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PADD (Guimbretière, 2003)</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PaperProof (Weibel et al., 2008)</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Papiercraft (Liao et al., 2005, 2008)</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProofRite (Conroy et al., 2004)</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCA (Plimmer et al., 2006, Priest and Plimmer, 2006, Chang et al., 2008)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
### TABLE 2.9: (continued).

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Group</th>
<th>Recognise</th>
<th>Anchor</th>
<th>Store</th>
<th>Reposition</th>
<th>Refit</th>
<th>Orphan</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScreenCrayons (Olsen et al., 2004)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Steimle (2009) (Steimle, 2009)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>u-Annotate (Chatti et al., 2006)</td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>United slates (Chen et al., 2012, 2013)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>vsInk (Sutherland and Plimmer, 2013)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Wu, Yang &amp; Su (2008) (Wu et al., 2008)</td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

Number of Implementations: 7 18 16 13 9 4 4

### 2.2.5 User Studies

The user studies are divided into three types:

(i) Usability
(ii) Technical capacities
(iii) User expectations

Usability studies investigate the effectiveness and efficiency of the implementation. There are a number of usability studies that looked at the usefulness and
learnability of various implementations (Anderson et al., 2005, Golovchinsky and Denoue, 2002, Marinai, 2013, Mazzei et al., 2013, Steimle, 2012, Asai and Yamana, 2014). These studies have identified a range of issues that need to be considered. However, few of these studies took their results and generalised them beyond the implementation. This makes these results specific to the implementation itself without exploring the wider possibilities of what it means for user expectation.

Studies on the technical capacities investigate the technical limitations of the implementation. These might include speed, performance and accuracy. Again these studies are limited to the implementation; although the details do often show where the implementation can be improved.

The final type of study is on user expectations. These investigate new avenues that are not possible with paper-based annotations. XLibris is an example of an implementation that was used to investigate user expectations in a variety of contexts (Price et al., 1998a, Schilit et al., 1998a, Golovchinsky et al., 1999, Marshall et al., 1999, 2001, Shipman et al., 2003). While these studies are interesting and provide more detail on user expectations for this review the specific focus on user expectations is the automatic adaptation of annotations. There is only one study in this area (Bargeron and Moscovich, 2003).

One important finding is that users like implementations that are predictable and reliable. However, if this is not possible then they do not want the implementation to change their annotations. Bargeron and Moscovich (2003) found users would prefer the underlying text to be locked, so they cannot modify it if the annotation cannot be accurately adapted. They theorised there would be a cut-over point for when to lock the context but they were unable to detect one based on their results.

Another important finding is people are happy with “cleaned” annotations. These annotations are often preferred over the original annotations. In addition, people are happier seeing these annotations change in response to changes in the underlying document than seeing their original annotations change. However, cleaning the annotations increases user expectations. The users have a higher expectation that the implementation understands their meaning (Bargeron and Moscovich, 2003).

While only one study specifically looked at user expectations for adaptation there are several other studies that include results related to automatic adaptation.
One reason why annotations do not always behave as expected is because of the grouping of strokes into an annotation. Some annotations (e.g. text, drawings, etc.) are expected to remain together. For example, the cross stroke of the ‘t’ and the dot above the ‘i’ should remain in position relative to the rest of the letter. In some automatic implementations, these strokes separate during repositioning (Golovchinsky and Denoue, 2002, Marinai, 2013).

Another area that can cause confusion is resizing multiple line annotations (Bargeron and Moscovich, 2003, Golovchinsky and Denoue, 2002). When the annotation is outside the text this is not an issue (e.g. margin bars or braces) but when the annotation is within the text the meaning is not preserved as effectively. One potential reason for this is adaptations for this category of annotation do not take into account which words the annotation should be associated with.

Anderson et al. (2005) suggest that digital annotation can be more difficult to read than annotations on paper. Identified factors that cause this include:

(i) Pen size: often the pen is a larger size than would be used on paper. The annotations can take up too much space on the document and obscure the underlying text;
(ii) Pen colour: the colours chosen for the annotations can make them more difficult to read (especially when displayed via a projector);
(iii) Annotation similarity: all annotations added using digital ink have the same colour. Unless the user changes the colour, all ink annotations in the same location will merge together.

These factors make it harder for people to accurately add annotations to the document. This reduction in accuracy then has a flow-on effect where annotations are more difficult to correctly adapt. DIZI is an example of a system that attempts to overcome these challenges (Agrawala and Shilman, 2005).

Anderson et al. (2005) further suggest the last point occurs because the digital ink doesn’t change over time like pen annotations do. They claim when ink is first added to paper it appears slightly different. Then as time passes the ink dries and the colour changes slightly. This makes it easier to differentiate annotations based on the time they were added. The colour also changes when multiple strokes are layered on top of each other with a pen but with digital ink, all the strokes have exactly the same colour.
2.3 Discussion

During this review, a number of areas of significance were found. This section discusses these and how they impact on freeform digital annotations. Three adapting operations reported were identified: repositioning, refitting and orphaning. Before these operations can be applied there are four adding operations that influence automatic adaptation: grouping, recognising, anchoring and storing. The effectiveness of the adapting operations depends on the effectiveness of these adding operations. Therefore §2.3.1 discusses the adding operations followed by §2.3.2 on the adapting operations. One important factor that influences adaptation is the type of annotation. Previous work has identified different types of annotations but these are normally limited to static documents. §2.3.3 describes how the taxonomy handles dynamic documents and use this to identify areas that have not been covered in the current literature. Another important concept that influences adaptation is annotation lifetime; §2.3.4 discusses this and how it interacts with fluidity. §2.3.5 addresses the lack of research on user experience and going from individual usability studies to the wider picture. Finally, §2.3.6 discusses the results for Integrated Development Environments.

2.3.1 Adding Operations

Grouping is the process of combining multiple gestures together into a single annotation. This is required because people do not think of annotations as a series of individual gestures but as a single group. In contrast, computers process the digital ink as individual gestures. When the gestures are grouped together successfully then the entire annotation appears as a single unit. One common problem with adapting an annotation is when individual parts of the annotation move independently of the others (Golovchinsky and Denoue, 2002, Marinai, 2013). This then causes confusion as the user expected the whole to stay together.

Recognition is the process of understanding either part or all of the annotation. This is important because different annotation types require different actions (Golovchinsky and Denoue, 2002, Bargeron and Moscovich, 2003). For example, adapting an underline requires the underline stay underneath the associated words. If the annotation is incorrectly recognized, then the wrong action will be applied to it, again resulting in user confusion. Another potential area of
investigation that relies on recognition is cleaning annotations (Bargeron and Moscovich, 2003). Cleaned annotations potentially provide an intermediate representation that is easy for processing but still remains understandable to the user. However, they must be correctly recognised to remain understandable as incorrect recognition would result in a wrong cleaning operation being applied.

Anchoring is the processing of associating the annotation with a location in the underlying document. Anchoring is a key prerequisite for repositioning: the annotation will only move to the correct location if the anchor is correct. However, as identified in the literature, there are a variety of different anchoring approaches. The simplest approach is to associate the annotation with the page. This only requires a graphical offset to the top of the page without any need to identify individual elements on the page. But what this gains in simplicity it loses in functionality: there is no way to do any adapting operations at a lower level than the page. In contrast, the most granular level is to identify individual words (or even letters). But this approach raises more issues: first, how to correctly identify the anchor word; second, how to find this word again after the document has changed; and third, how to adapt the annotation if it spans multiple words. Other approaches have used less granular anchors (line or paragraph level) or anchors based on the underlying document code (e.g. HTML). What is common to all these approaches is there is a trade-off. Moving to a more granular level allows more control and flexibility but at an increasing cost of complexity. As yet, there is no clearly identified “best” approach. Instead, it depends on what the implementation is trying to do and how important it is to accurately adapt the annotation.

The final operation, storage, is less important for adapting annotations. Its impact is how annotations can be retrieved later. For implementations that are only interested in immediately evaluating the adding or adapting operations, this operation can be omitted altogether. However, for implementations that need to be persisted this operation is important. One current issue with this operation is the data that is stored. There is not a common data format or storage location; thus, each implementation needs to implement this operation itself. There are some common formats (e.g. InkML or Microsoft’s ISF format) but these appear to be focused at the digital ink level rather than the higher level of annotations. Thus, using these formats requires extensions to include relevant information.
Both anchoring and storing have well-studied solutions. There are sound solutions for anchoring that work in most circumstances. While storing uses a variety of datatypes this is not a fundamental concern. Recognising and grouping are both related and challenging. The problems encountered with recognition and grouping are being investigated in the wider field of freeform digital ink. A comprehensive review of recognition and grouping in the sketching domain was reported in (Johnson et al., 2009). The interested reader is referred to this work for further details.

In addition to the individual steps, there is also the question of what order should these steps be applied. Two general sequences were identified: process each annotation individually (one-off); and process all annotations each time a change is made (recurrent). Each approach offers benefits and trade-offs. One-off processing is faster but information from other annotations can help with processing an annotation. It also fixes the annotation type at the time it is added; whereas for recurrent processing, the type may change as other strokes are added. Thus one-off processing allows stability for the user while recurrent processing may potentially be more accurate. This is an open area of research.

One area of recognition that is lacking is how well recognition works in an Integrated Development Environment. However, before this can be investigated we need to know what sort of annotation types might be used. If the types are different from previous studies, then we would need to investigate how well-existing recognisers work for these types. A second related issue is how the order of processing may affect recognition. The current studies that report accuracy rates all use a recurrent approach and it is unknown how accurate a one-off approach might be for annotations.

2.3.2 Adapting Operations

The current literature has mixed results about the adapting operations. Repositioning by itself has well-defined solutions. The errors with repositioning are not due to the repositioning itself but because of errors in the adding operations. Improving the adding operations, either individually or together, will improve repositioning without any additional work. In contrast, both refitting and orphaning do require additional work. Also, these two areas are under-represented in the research. Refitting has only been implemented in four implementations and the focus has been on a very narrow set of annotation types (single line
and multiple line). Of these four implementations, only one has looked at user expectations. Orphaning has also been in four implementations and only one type of orphaning has been implemented. In addition, none of these have looked at user expectations. In the aligned field of text annotation, studies have found that users expect annotations to be available after their anchor has been deleted (Brush et al., 2001, Sanderson and Van de Sompel, 2010). Based on prior research this could take two forms. The first is to store all the orphaned annotations so the user can review them later. An alternate approach would be to show an icon at the "best guess" location on the associated document (Brush et al., 2001). Selecting this icon would then display the annotation. For each approach, the user should be provided options on what to do with the orphaned annotation (e.g. delete, reposition, modify).

Thus, there are well-studied solutions for repositioning freeform annotations but both refitting and orphaning require additional investigations. The solutions for repositioning work in expected ways that maintain the meaning of the annotations. There is a robust solution for refitting single line annotations but not the other categories of annotations. The only approach for orphaning previously studied is deletion and there is no work investigating alternate approaches.

2.3.3 Annotation Categories

In her original taxonomy, Marshall (1997) classified annotations along two dimensions: within-text vs. marginal or blank space; and telegraphic vs. explicit. During her investigation, Marshall looked at annotations added to textbooks - one assumes for student study. Both the annotations and their underlying context are static. In contrast, digital documents are dynamic so the content underneath annotations can change. Thus, Marshall’s taxonomy, while still valid, is limited to static documents. Some annotations on dynamic documents do not fit in Marshall’s four quadrants. For example, connector annotations for both within-text and marginal and commands may be added anywhere in the document. Given these challenges, an alternate taxonomy is defined here based on how the annotation would adapt in a digital environment.

This taxonomy builds on the work by Golovchinsky and Denoue (2002) and Bargeron and Moscovich (2003). In their work they grouped annotations into

---

8Prior work indicates the associated context must also be stored (Sutherland and Plimmer, 2013).
three categories: single line, multiple line and complex\(^9\). Previously both connectors and commands were categorised as complex annotations. This is partly because these categories were used as a basis for automatically adapting annotations. Single line and multiple annotations were refitted as the underlying content changed while complex annotations were not. To these three categories connectors and commands are added (see Table 2.5).

Connectors are often used to link another annotation to a location in the underlying document (Priest and Plimmer, 2006, Chen and Plimmer, 2007, Wang and Raghupathy, 2007). Accordingly, they potentially have two anchor points: one fixed to a location in the document and another associated with an annotation. An alternate form of connector is one that joins two sections of the content. These connectors also have two anchor points but both associated with the document. There has been no research published investigating how to automatically adapt connectors. Connectors should be repositioned like any other annotation; I also speculate they would follow similar rules for orphaning. However, refitting is a more interesting scenario: theoretically, it would be possible to refit connectors so the two anchor points move independently. There has been no research published on this work or, more importantly, on what the user expectations are. I postulate that there are two forms of refitting a connector. For connectors associated with another annotation, the connector and associated annotation should be repositioned so the document anchor remains valid. For connectors associated with two different locations in the document, the connector should be stretched or shrunk so the two anchor points remain in the correct locations.

Commands are often used for instructing the implementation to do something (e.g. erase or move content (Weibel et al., 2008, Hardock et al., 1993, Conroy et al., 2004), move to another location (Anderson et al., 2004a,b, 2005, Truong et al., 1999), link documents (Steimle, 2009), etc.) Unlike most other annotations, the implementation is expected to understand these annotations. One major area of research for commands is how to recognise them (see below). Unlike most other forms of annotation, commands are transient and have only a short-term lifetime. Again, there has been no research on automatically adapting command annotations. I also posit these would follow the same rules for repositioning and orphaning as other annotations. However, given that commands are temporary, the value of refitting them is questionable.

---

\(^9\)Both publications included similar annotation types in each category. The only exception is enclosures. Bargeron and Moscovich (2003) treated these primarily as single line annotations while Golovchinsky and Denoue (2002) treated them as multiple line.
Combining the two taxonomies (annotation types and annotation support operations), some areas for future research are refitting connector, command and complex annotations, and orphaning. There are very few implementations that look at the technical complexities and none that investigates user expectations.

2.3.4 Annotation Lifetime

The lifetime of an annotation is an important concept that, while evident from this review, is not widely discussed. There is a continuum of lifetimes; with three major points on the continuum: instantaneous, short term, and long term. Functional commands are instantaneous. The command is executed and the annotation discarded. Short-term annotations have a limited lifetime. Once the annotation indicating that something needing editing has fulfilled its purpose it is removed. Long-term annotations become part of the document; for example, providing commentary or explanatory notes.

There is an interplay between lifetime and fluidity. In order to differentiate functional commands and digital ink, many systems required the user to change modes (which is cognitively disruptive). This is because the software has difficulty reliably differentiating gesture classes. There is ongoing research into gesture recognition that may provide a solution; currently there is a need to provide the software with a way to separate commands from ink. There are a number of solutions suggested including using buttons (pen-based or separate), separate display areas, special gestures, pressure and pen plus touch. Each of these has its own limitations and strengths: which is most suitable is context dependent. Buttons require specialised hardware and added user dexterity, but have the advantage of certainty. Separate display areas use screen real estate and require a move in focus for the user but can provide a zoomed area for writing. Both special gestures and pressure require training (the user, the system or both) and recognition errors are still possible but they result in a more fluid interaction. Pen plus touch requires special hardware, bimanual interaction and recognition but has the most potential for providing fluid interaction and builds on human bimanual abilities: for example, a person may draw with pencil in one hand and an eraser in the other. Li et al. (2005) investigated the performance of some of these approaches and found a bimanual approach (pen in preferred hand and button-push with non-preferred hand) was the fastest. This approach also had one of the lowest error rates and was preferred by most participants.
2.3.5 User Experience

Of note is that the research into the technical issues with annotation is more advanced than the research on user experience. While the work of Marshall (1997) laid an excellent foundation for how people annotate books, many of the studies reported here focus entirely on the technical issues. Those user studies that are reported (for example Olsen et al., 2004, Mackay et al., 2002, Sutherland and Plimmer, 2013), are usually usability studies that evaluate the usability of the specific application without regard to the fundamental and theoretical principles. Only two studies (Golovchinsky and Denoue, 2002, Bargeron and Moscovich, 2003) investigated user expectations on adapting annotations. While there has been work in this area for text-based annotation there is an urgent need for more work in this regard.

In addition, most studies have focused on evaluating the effectiveness of their own implementation. Very few implementations attempt to generalise beyond their initial implementation\(^\text{10}\). While there are many solutions to technical challenges and interesting ideas for functionality, it is hard to generalise beyond the initial implementations. What works in one particular implementation, with its specific environment and objectives, may not work when transposed to another implementation. This may be a limitation of the field, where we focus on smaller units of work rather than exploring the bigger picture, that limits the transferability of our findings outside our field. This raises the question: Are we ignoring the bigger picture of how our work could benefit mankind and leaving it to industry? This is a serious issue as industry has different objectives and driving motives which skew the long-term benefit of our research.

2.3.6 Integrated Development Environment Integration

Prior studies have shown that it is possible to integrate ink into an Integrated Development Environment. Visual Studio 2010, and following, allow ink to be directly integrated into the code editor, which overcomes one of the major limitations in earlier studies (Chang et al., 2008). Now that it is possible to integrate ink, the question becomes: What is the value of adding freeform ink? The studies mentioned hypothesise that freeform ink is easier and less mentally demanding than using a keyboard and mouse to add annotations (Chen and

\(^{10}\)XLibris is the main exception.
Plimmer, 2007, Sutherland and Plimmer, 2013). But to date, there are no studies that confirm this hypothesis.

The studies identified two avenues of research for freeform digital ink. First, digital ink may be useful for navigation (Chen and Plimmer, 2007, Lichtschlag et al., 2014). Whether this would replace the in-built navigation tools or provide an ancillary approach is unknown. Second, being able to handle changes to the code is important. Program code is dynamic, with people often reading code because it has been changed or is about to be changed. Should the underlying code change, and the annotation not adapt, then the meaning of the annotation would be lost. Previous studies in other areas have shown users want their annotations to retain meaning (Bargeron and Moscovich, 2003, Marinai, 2013). Nonetheless, there have been very few studies to investigate how users expect their annotations to adapt, for code in other contexts.

2.4 Summary

The motivation for this review was to determine what has been investigated for freeform digital ink annotations on text documents. Four research questions were formulated to guide the review:

**MS.Q1:** What operations are needed for adding and adapting annotations on dynamic, digital documents?

**MS.Q2:** What types of annotations need to be handled in an annotation application?

**MS.Q3:** What are the user expectations for annotation adaptation?

**MS.Q4:** What has been implemented for Integrated Development Environment integration in the area of freeform annotations?

A systematic mapping study was performed to review the current literature and present a taxonomy of current work.

Adding annotations to documents is well covered in the research. There are four operations used for adding annotations: grouping, recognising, anchoring, and storing. However, there is not a common order to how these operations are used; instead, there are variations based on the overall approach used and the level of user interaction provided. Ten commonly recognised types of annotation were identified and grouped into five categories based on their requirements for adding and adapting.
Automatic adapting of annotations has not been investigated widely. Repositioning is the most common adapting implementation, followed by orphaning and then refitting annotations. If the annotation is added robustly then repositioning occurs without additional work. The implementations that implement orphaning, all work by deleting the annotation. The two implementations that refit annotations only look at a reduced set of annotations: single-line and simple multi-line annotations. This is an area that needs additional investigation.

The most common type of human study is a usability study of how well an implementation performs but these do not improve the overall understanding of the underlying user expectations. There are few studies that investigate user expectations and only one that studied adapting annotations. This is a major limitation in the current literature. The review did not reveal how people will react to different types of changes or even how they might want an annotation to change. There are also problems in that certain forms of annotation are not adapted by current systems. Future work in digital ink annotation research should use the taxonomy presented here to describe how the research relates to the field.

Finally, previous studies have shown it is possible to integrate freeform digital ink into Integrated Development Environments. Modern Integrated Development Environments, at least Visual Studio, allow integrating ink directly within the code editor itself. While it is possible to integrate digital ink directly within an Integrated Development Environment, integration has only been implemented as proof-of-concept research projects. Currently, there are no commercial implementations that allow freeform digital ink annotations. A wider question is: What value would ink annotations on code provide? One potential area is navigation yet it is unknown how this would merge with existing navigation patterns. A second question is: How should annotations adapt to changes in the code? This question is relevant to both annotating code and to annotating other forms of text.

In this chapter, three areas were identified that are of relevance to this thesis. The first area is how programmers might use annotations in an Integrated Development Environment: this is the focus of the first user study (see Chapter 3). The second area is how to automatically recognise annotations in an Integrated Development Environment: this is the focus of the second user study (see Chapter 4). The final area is how can annotations be automatically refitted: this is the focus of the third user study (see Chapter 5).
This chapter identified the areas of research for the remainder of the thesis. The next chapter will describe the first iteration in the thesis. This iteration explored how and why programmers annotate program code and provided background information on how annotations could be implemented in an Integrated Development Environment.
CHAPTER 3

User Study 1
Honey, I’ve Lost the Keyboard

“Science, my lad, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.”

Jules Verne

The previous chapter described the overall research methodology. This included a high-level overview of the three iterations and the research questions addressed for each. This chapter addresses the first of the research questions: “How and why do programmers annotate programming code on paper?”

This chapter describes the user study conducted to understand how and why programmers use annotations when asked to read and understand some program code. Section 3.1 describes the aim of the study and the rationale for undertaking a paper-based study. Section 3.2 describes the design of the study: the experimental task, the participants and how the data was analysed. Section 3.3 describes the main findings from the study. This includes the types of annotations added, why they were added and what participants would like in an annotation system for program code. Section 3.4 discusses these findings in the context of this thesis and how they inform the design of an annotation system for program code comprehension, then Section 3.5 discusses some of the limitations. Finally, Section 3.6 closes this chapter with a summary.
The study described in this chapter was conducted with the approval of the University of Auckland Human Participants Ethics Committee. A copy of the approval letter is in Appendix A.

Some of the results from this study have been published previously in Sutherland et al. (2015).

### 3.1 Purpose of this Study

As identified in the literature review in Chapter 2, it is technically possible to add freeform digital annotations in an Integrated Development Environment. What prior research has not identified is whether this functionality would be useful. This chapter explores this area and investigates what sort of functionality would be required.

There is already evidence that annotations are an important part of reading (e.g. O’Hara and Sellen, 1997, Adler et al., 1998, Tashman and Edwards, 2011b). Annotations can help with remembering information (Simpson and Nist, 1990), finding previous locations (Crisp and Johnson, 2007) and reformulating problems (Jackel, 2014). As these are all tasks performed during code comprehension (Sillito et al., 2005, Ko et al., 2006) it would stand to reason that annotating would also be beneficial for understanding program code.

One theory used to explain how annotations may assist with comprehension is the Theory of External Cognition (Scaife and Rogers, 1996a,b). In the Theory of External Cognition, there is an interaction between internal and external representations of information during a cognitive task (Rogers, 2004). External representations allow for computational offloading: computational offloading reduces the amount of cognitive effort required in a task (Rogers, 2004). Three main forms of computational offloading have been proposed (Scaife and Rogers, 1996b):

**Re-representation**: the problem or task is presented in a different format but with the same underlying structure and concepts.

**Graphical constraining**: the graphical elements used constrain the kinds of inferences that can be made about the underlying concepts.

**Temporal and spatial constraining**: different elements of the problem are more salient when distributed over time and space.
At the same time, not all annotations are equal. Freeform hand-written annotations are more beneficial and preferred than typed annotations (O’Hara and Sellen, 1997, Morris et al., 2007). Anecdotal evidence points to a shift away from printing out program code, especially as one moves beyond learning. Programmers are becoming more constrained to the tools available in Integrated Development Environments and Integrated Development Environments are typically text-based environments with some graphical tools (Maalej et al., 2014). As Integrated Development Environments and their embedded tools typically use a keyboard and mouse as the input mechanism, we have potentially lost a tool for program code comprehension.

Currently, Integrated Development Environments provide a wide variety of tools, some of which have become almost indispensable for code comprehension (Maalej et al., 2014). Therefore, the tools in Integrated Development Environments may provide some alternate tools that are just as useful as annotations. Research has not yet shown where the overlap is between these tools and what annotating provides. Potentially, some tasks may be possible in both Integrated Development Environment tools and with annotating, while other tasks may only be possible in one or the other. Before building an annotation tool for an Integrated Development Environment we need to identify where the value is in building such a tool.

This study investigates how and why programmers annotate program code on paper. The following hypotheses were proposed:

H1: Programmers will still annotate the code in a similar way to annotating other forms of text.

H2: The main reasons for annotating would be to reduce the reader’s mental workload (computational offloading). Problem reformulation together with temporal and spatial constraining would be the main approaches used.

In addition, the following sub-questions were asked:

1(a) Do programmers annotate on program code?
1(b) What are the types of annotations added?
1(c) How do these types differ from previously reported studies?
1(d) What are the main reasons for adding any annotations?

Given these research questions, the next section describes the study design used to answer them.
3.2 Study Design

The previous section outlined why it is important to investigate how and why programmers annotate on program code. This section describes the design of the study, the participants, the task and how it was applied, and how the data was analysed.

3.2.1 Methodology

This study involved observing experienced programmers reading a program they had not seen before. The participants were all given code printed on paper. This was to reduce any confounding effects due to prior experience with a specific Integrated Development Environment. In addition, there are very few tools that allow freeform digital ink annotations in Integrated Development Environments and none are commercially available. While there was a prototype of vsInk available this may have influenced the results of the study because the participants would need to be shown how vsInk works. Without any explanation, the participants would not be aware they can freely annotate on the code. This may have conveyed to the participants they would need to add annotations; whether they would normally or not.

The protocol called for deceiving the participants. At the start of the study, the participants were told the purpose of the study was to investigate how they went about understanding code. This way the participants were focused on their normal processes for understanding code rather than focusing on adding annotations. As mentioned above, if the participants were aware of the focus on annotations they might add them, potentially introducing bias into the study.

Finally, experienced programmers were chosen as the participants because the target audience is experienced programmers. With experienced programmers, their focus would be understanding the code rather than learning the programming language.
3.2.2 Participants

There were thirteen participants in this study. All were programmers with at least five years programming experience. They all stated they had prior experience with C#. All participants either currently worked in a commercial environment or had previously worked in one. All but one participant were male.

3.2.3 Task

The participants were asked to read a short block of program code. Prior to the reading task, they were told that they would have to explain how the code works to a less experienced programmer. The rationale given for the task was that a new graduate would be taking over responsibility for the code and needed assistance understanding the code. They were not told to annotate the code. If they asked, they were told they could do anything with the paper; including drawing or writing on it.

The code consisted of six files from a larger system written entirely in C#. The code in this study was responsible for initializing communications between modules in the system. All relevant code was provided. The code was printed for the participants to read. The printed code used 14 pages of paper. The code was printed from Visual Studio using the following settings:

- 2.5cm margins
- Black and white
- Single sided
- Line Numbers

Each participant was given a copy of the code, some extra blank pieces of paper, a highlighter and four different coloured pens (red, blue, black and green).

The participant was seated at one side of a table. At the other side of the table, a video camera was set up to record what the participant was doing. The camera was focused on the participant’s hands and the pieces of paper. The investigator sat behind the camera and ensured it was correctly focused and adjusted as needed during the task. The investigator also took notes while the participant was reading the code.
3.2.4 Procedure

Each participant performed the task in a controlled environment. Prior to the reading task, each participant was welcomed to the lab, had the process explained to them (including gaining consent) and filled in a short questionnaire on their programming and reading background.

Each participant was given 45 minutes to read the program code. They could finish earlier if desired. The fastest participant took 23 minutes to finish and four participants were stopped at 45 minutes. The average reading time was 38 minutes. While reading each participant was video recorded and observed. During the reading, questions about the process were answered (e.g. can I write on this paper?) but not about the code (e.g. what does this class do?). At the end of the reading time, the pieces of paper were collected and the purpose of the study was explained.

After the reading task, there was a second questionnaire and a short interview about their annotations. In the first part of the interview, the participants were asked about their general annotation and code reading practices. This was to gather background information to help understand why they annotated while reading. The section involved looking at the annotated paper and asking the participant to explain the type of the annotation and why they had made each annotation. This included the importance of each annotation and how it fitted into their approach to understanding the code. Finally, we talked about how they might use freeform annotations in an Integrated Development Environment.

3.2.5 Analysis

After each participant finished the data was coded. Each page was reviewed: the number of annotations counted and the following attributes recorded about each:

- Location
- Type of annotation
- Reason for adding annotation

Location was either within code or in the margin. If the annotation was in the margin, then the margin was also recorded (left, right, top, or bottom). This was based on where most of the annotation was.
Type of annotation was based initially on the annotations described by Marshall (1998). When a potentially new type was found the list was expanded to include it. After coding, the list was reviewed and similar types of annotations were combined.

During the interview, the investigator asked the participant the general purpose of each annotation. This was then summed up in a few words (e.g. “question”, “highlight for later”, “possible bug”, etc.) The reasons were then consolidated into a list of basic reasons based on the participant’s intention when the annotation was added.

These attributes were then summarised and collated with the answers from the questionnaires.

3.3 Findings

This section describes the main findings from the study. The first subsection contains the background details about the participants, the second subsection describes the annotations the participants made, the third subsection describes why they annotated, the fourth subsection describes the participants’ feedback on what would be useful in an annotation system, and the fifth subsection describes some incidental findings of interest. Finally, the sixth sub-section returns to the research questions for this study.

3.3.1 General Background

Most of the participants currently used C#; the remainder of the participants had used it previously and were confident in being able to read a program in it. The participants who currently used C# reported using it daily.

All participants had at least five years’ experience (see Table 3.1). All participants reported themselves as confident or expert in reading and understanding program code. Eleven of the participants reported reading their own code on a daily basis. The remaining two participants very rarely read their own code; however, they did read other people’s code on at least a weekly basis. Nine of the participants read other people’s code on a daily basis; three read on a weekly basis and only one participant rarely read other people’s code. This shows the participants were all familiar with reading code regularly.
Twelve participants read code on screen daily; the final participant would read it at least weekly. Only one participant reported reading code on paper regularly; the other participants reported rarely reading code on paper. These results match how often participants print out code.

Reasons for reading included debugging, learning, reviewing other people’s code and general development (e.g. extending an existing codebase).

During the study twelve of the participants added annotations or notes. Only one participant (P06) did not add any annotations or notes. In the post-observation interview this participant stated he “…never make any annotations and very rarely take notes when reading”. He also added, “This is how I was taught when I studied computer programming and I never needed to change.”

### 3.3.2 Annotation Details

In total, the participants added 267 annotations (see Table 3.2). Ten participants added annotations on the code and nine participants added annotations on separate pieces of paper. Two participants who did not annotate on the code did write on separate pieces of paper; the final participant did not make any annotations or notes.

<table>
<thead>
<tr>
<th>Participant #</th>
<th>Occupation</th>
<th>Years Writing Code</th>
<th>Reported Expertise</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Software Developer</td>
<td>10+</td>
<td>Confident</td>
</tr>
<tr>
<td>2</td>
<td>Software Developer</td>
<td>2-5</td>
<td>Confident</td>
</tr>
<tr>
<td>3</td>
<td>PhD Candidate</td>
<td>7-10</td>
<td>Confident</td>
</tr>
<tr>
<td>4</td>
<td>MSc Student</td>
<td>7-10</td>
<td>Confident</td>
</tr>
<tr>
<td>5</td>
<td>Lecturer</td>
<td>10+</td>
<td>Confident</td>
</tr>
<tr>
<td>6</td>
<td>PhD Candidate</td>
<td>10+</td>
<td>Expert</td>
</tr>
<tr>
<td>7</td>
<td>Software Developer</td>
<td>5-7</td>
<td>Confident</td>
</tr>
<tr>
<td>8</td>
<td>Software Developer</td>
<td>7-10</td>
<td>Expert</td>
</tr>
<tr>
<td>9</td>
<td>Software Developer</td>
<td>10+</td>
<td>Expert</td>
</tr>
<tr>
<td>10</td>
<td>Software Developer</td>
<td>5-7</td>
<td>Confident</td>
</tr>
<tr>
<td>11</td>
<td>Software Developer</td>
<td>7-10</td>
<td>Confident</td>
</tr>
<tr>
<td>12</td>
<td>Software Developer</td>
<td>7-10</td>
<td>Confident</td>
</tr>
<tr>
<td>13</td>
<td>PhD Candidate</td>
<td>10+</td>
<td>Expert</td>
</tr>
</tbody>
</table>
For those participants who annotated the code, the average number of annotations added was 27. The number of files annotated ranged from one to all six.

The following types of annotations were identified:

- **Underline**: a line drawn underneath text
- **Scratch-out**: a line drawn through text
- **Highlight**: a line drawn through text with the highlighter
- **Enclosure**: a circled block of code
- **Margin bar**: a vertical line, typically drawn in the margins
- **Brace**: a { like annotation spanning multiple lines of code
- **Connector**: a joining line between two elements (e.g. a block of code or another annotation) without an arrow
- **Arrow**: as above but with an arrow head
- **Text**: one or more characters
- **Drawing**: a diagram or other drawing, often with explicit meaning
- **Dot**: a small mark on the paper, often incidental

These annotations were classified into five categories. Three of these categories (single line, multiple and complex) were also identified in previous work (Golovchinsky and Denoue, 2002, Bargeron and Moscovich, 2003):

- **Single line**: these span a single line and are associated with the code or other text. Underlines, highlights and scratch-outs all fall into this category.
• **Multiple line**: these span multiple lines of code but do not contain explicit meaning beyond their location. Enclosures, margin bars and braces fall into this category.

• **Connector**: these join two or more items together. One common use for a connection annotation is to associate a text annotation with a segment of code. Arrows and connectors fall into this category.

• **Complex**: these have explicit meaning associated with them, although maybe only to the original annotator. Text and drawings are both in this category.

• **Attentional**: these are indirect signs of the participant’s attention. They are temporary and not intended for future re-use. Dots fall into this category.

In addition, some annotations can be split into sub-annotations of different categories. These annotations were classified as *compound* annotations. As the participants added them at the same time, they considered the whole annotation as one annotation, rather than separate smaller annotations. A common example is a connector and text together. In the analysis, both the sub-annotations and the whole annotation were included in the category totals. For an annotation to be considered a compound annotation, the sub-annotations must be of different categories.

The annotations added by each participant are shown in Figure 3.1. While there are no common patterns among the participants in this study there are two general trends. Four of the participants (P01, P03, P04 and P07) preferred mainly single line annotations. Four of the remaining participants (P02, P09, P11 and P12) preferred mainly complex annotations. These annotations are predominately text notes.

Most of the annotations were added within the code (65%). The next most common location is to the right of the code (24%) and then to the left or top of the code (6% and 5%). Only one participant added an annotation below the code – this involved copying some lines of code from the next page. These results are based on code printed from Visual Studio using the default margins (2.5cm per margin). However, the right margin was jagged (due to the nature of the code) which resulted in considerable additional whitespace. Figure 3.2 shows where each participant added the annotations.

There is a relationship between the classification of annotation and its location. Single line and attentional annotations mainly occurred within the code. Multiple line and connector annotations mainly occurred to the left or the right of the
Figure 3.1: Percentages of annotation types added on the source by classification per participant. Percentages are the percentage of total annotations added by the participant. Compound annotations are a separate classification – therefore the totals may add to more than 100%.

Figure 3.2: Percentage of annotations added to each location per participant. Percentages are the percentage of total annotations added by the participant.
code. Complex annotations occurred at any location.

3.3.3 Reasons for Annotations

Initial data entry identified 39 different reasons for adding the annotations. During the analysis, many reasons were found to be similar in nature; therefore similar reasons were grouped together resulting in eleven basic reasons (see Table 3.3).

These basic reasons were grouped into four categories based on the main intent. The first category, navigation, is for helping the participant find their way through the code. The participants claimed these annotations speed up the process of re-finding information. The second category, working information, is where the participant is recording things they have found about the code. This is information they think will be useful later. The third category, information for sharing, is similar but intended for someone else. The final category, other, is for the more uncommon reasons.

Navigation Annotations

The most common reason for adding an annotation is to emphasise for future. This marks an item the participant is thinking of returning to later – even if they never do. Common examples of this are: underlining a method call or name, circling a section of interesting code and highlighting where variables are defined. These annotations were generally single line annotations. Figure 3.3 shows examples of this sort of navigation with the highlighted class and method names.

There were two participants who extensively annotated for this reason (P01 and P04). They both said they were trying to build a model of the code. P01 said he was trying to replicate the solution explorer in Visual Studio (the solution explorer lists all the classes and methods in a set of code files). He went quickly through the code initially to highlight most of the method names. Then he went through the code a second time using the highlighted method names to trace what was happening. In contrast, P04 used the highlighted method names as a form of backtracking. When he started in a method he would highlight the method name. Later he could easily return to this method by scanning through the paper for the highlight.
Table 3.3: Consolidated list of reasons why annotations were added

<table>
<thead>
<tr>
<th>Reason</th>
<th>Description</th>
<th>Number</th>
<th>Percentage¹</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Navigation</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Emphasise for future</td>
<td>The annotation marks a feature of the code to be reviewed later</td>
<td>117</td>
<td>44%</td>
</tr>
<tr>
<td>Add reference</td>
<td>A reference to a different section in the code</td>
<td>21</td>
<td>8%</td>
</tr>
<tr>
<td>Emphasise code structure</td>
<td>Highlight a structural element in the code</td>
<td>17</td>
<td>6%</td>
</tr>
<tr>
<td>Emphasise significant feature</td>
<td>A section of the code that needs further investigation</td>
<td>9</td>
<td>3%</td>
</tr>
<tr>
<td><strong>Working Information</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Record working notes</td>
<td>Inline notes on what is happening in the program</td>
<td>36</td>
<td>13%</td>
</tr>
<tr>
<td>Record question</td>
<td>A query about something in the code</td>
<td>36</td>
<td>13%</td>
</tr>
<tr>
<td>Correct previous annotation</td>
<td>A correction or update to a previous annotation</td>
<td>6</td>
<td>2%</td>
</tr>
<tr>
<td><strong>Information Sharing</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Record a needed change</td>
<td>An area of the code that needs to be changed</td>
<td>11</td>
<td>4%</td>
</tr>
<tr>
<td>Emphasise example</td>
<td>An example of another section of code is called</td>
<td>2</td>
<td>1%</td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unknown</td>
<td>The participant was unsure as to why they added the annotation</td>
<td>6</td>
<td>2%</td>
</tr>
<tr>
<td>Attentional</td>
<td>An incidental mark as a result of the user’s attention</td>
<td>6</td>
<td>2%</td>
</tr>
<tr>
<td><strong>Total Number of Annotations:</strong></td>
<td></td>
<td>267</td>
<td></td>
</tr>
</tbody>
</table>
Other participants who emphasised for future reference were more selective. They only emphasised the elements they were specifically interested in. There were two main motives: emphasising hard-to-find elements and commonly referenced elements. Both motives were to help the participant re-find them later.

Another reason is to add reference. These annotations link two or more sections of code together. The participants added these when they thought they would need to move quickly between two segments of the code. One example of this reason is matching letters to the side of the code. The participant was able to quickly scan through the code and find the matching reference as needed. These annotations were mainly text like a symbol or the name of the method. During the interview, participants said the references were not always used but they were valuable especially for hard-to-find segments of code. In Figure 3.3 (top), the underline and the circled text are examples of this reason.

For emphasising code structure the participant was trying to make it easier to move between different files. In all cases, the participants added these at the top of the first page in the file. These annotations were either text or a highlight. The highlight or text was always positioned so the participant could see it quickly.
Figure 3.4: Examples of annotations used for working information (top) and sharing information (bottom).
Emphasising a significant feature had two categories: important code and hard-to-find code. These differ from other navigation reasons in one significant aspect: the participant expects they will implicitly remember the rough location of the code. The annotation is there is to speed up finding it again. One participant mentioned this is like adding a post-it note to the page. The highlighting of the ClientRecord in Figure 3.3 (bottom) is an example of this reason.

Working Information Annotations

Three reasons for annotations were identified where they were for recording information: working notes; questions and corrections. All three reasons were an attempt to reduce the mental workload while reading. Figure 3.4 (top) shows examples of annotations added for these reasons.

Working notes are a description of what is happening in the code. The participant would add a working note when they figured out something and wanted to remember it later. The notes are a way of offloading this information onto paper to reduce their mental workload (Scaife and Rogers, 1996a, Rogers, 2004). These are typically text comments although they might only make sense to the writer. Occasionally a participant would return to their working notes and make a correction if they realised a comment was incorrect.

A question annotation is where the participant wanted to find out some more information. Questions are different from working notes because the participant is seeking an answer. In contrast, working notes are what the participant already understands. Unlike working notes, question annotations have a wide variety of forms. They could be as simple as an underline or asterisk next to some code up to a complete written question. During the interview, the participants provided more details about these question annotations including what the underlying question was and whether it was answered later.

Three types of question were found: implementation, functional requirements and language features. A question on the implementation is why the code was implemented a certain way. For example, one participant was interested in why there were two methods with the same name but different parameters. A question on the functional requirements was when the participant saw something that was not clear if it matched the described functionality of the system. For example, one participant questioned whether a method had the correct logic. Finally, a question on a language feature was when the participant saw something
they did not understand about the language or base libraries. For example, one participant queried about how a lambda was coded.

An implementation question was typically answered by the participant in their reading when they understood the code better. But only one participant updated these annotations with the answer. Most of the participants said they just remembered the information. The annotation reminded them of the question and answer. Questions about the functional requirements were sometimes answered when the participant read other parts of the code. None of the participants updated these annotations although some added a reference to the question. One participant described these as “questions to take to the business owner about whether the code is correct”. The final type were questions that would be answered by researching the feature: looking online, asking someone or looking up some form of reference materials. None of the participants updated these annotations during the session.

A correction annotation is where the participant has returned to a previous annotation and updated it. Corrections always involved scratching out part or all of an annotation. In two instances the participant also added additional alternate text.

**Information Sharing Annotations**

Information sharing annotations are similar to working information annotations. The main difference is information sharing annotations are intended for another person rather than the initial reader. As can be seen in Figure 3.4 (bottom), these annotations look very similar to the annotations for personal working information. The main difference between the two is the intent behind why they were added; geometrically they are the same.

Participants also added annotations to record a needed change. Examples of these annotations included hard-coded values, unclear variable or method names and bad coding practices. Most were enclosures; there was only one text annotation in this reason. The participants who added these annotations stated they only had to look at the code to remember why they added the annotation. This implies that these annotations are a form of offloading from memory onto the paper (Scaife and Rogers, 1996a,b).
The least common reason was *emphasising an example*. Both example annotations were where the participant thought the code was a very good example for a junior programmer to see.

Other Reasons

*Attentional* annotations are incidental marks of the participant’s current focus. During the task, the participants often pointed to the code with the pen currently in hand. Occasionally the participant would make contact with the page. During the interview, the participants who made these annotations mentioned they were accidental and they did not care about these annotations.

Finally, *unknown* was added: this is when the participant did not remember why the annotation was added. However, this was uncommon: they only failed to remember the reason six times out of 267 annotations. As can be seen in Figure 3.5, these annotations were typically a single symbol or mark on the page with no associated explicit information.

### 3.3.4 Participant Feedback

At the end of the study, the participants were invited to share what they thought of annotations within an Integrated Development Environment. This section summarises the suggestions and requirements.

Most of the participants thought being able to add annotations was a useful feature. Only two of the participants said they would never use this functionality. However, the participants did list some basic requirements before they would use it. The most mentioned requirements were:

- All the normal functionality in Visual Studio is still available
- The speed of Visual Studio is not impacted
- It must be possible to hide the annotations when desired
• It must be possible to edit annotations: both change them and remove them as needed

The suggestions from the participants were grouped into three main groups: navigation, collaboration/sharing and structure.

Navigation involves finding the annotations again in the future. The most common suggestion in this category is annotations should be searchable. The system should be able to understand the annotations and allow the user to search for the content within them (e.g. any text should be parsed and searchable). While several participants recognised the difficulty of parsing annotations they thought it should be possible to at least parse the text annotations and store this with the annotation. As an alternate, it should be possible to search for them using a variety of parameters, including type, date/time, person and file(s). A related suggestion is annotations should be filterable: the participant should be able to selectively hide/show annotations based on the same parameters. Finally, there should be some form of explorer that allows them to view all the annotations in a file and/or solution. Selecting the annotation should take the user directly to this annotation.

Collaboration/sharing suggestions are about making the annotations available to other people. As a minimum, multiple people should be able to add annotations to the code. Each annotation should be tagged as to who added it and when. Participants would also like to be able to mark annotations as private and/or modify annotations prior to sharing them. Finally, annotations should be stored with the code, both on the file system and any source code repository.

Structure involves the structure of the underlying code. One suggestion is if a method or variable was underlined then all instances of that item should be also be highlighted. Another suggestion is underlining a method or variable in the code should change it into a hyperlink to the definition of the item. The final suggestion is the annotations should be associated with the code structure (e.g. the Abstract Syntax Tree) and treated as another node in the structure.

3.3.5 Other Findings

In addition to the findings on annotations, there were two other findings that are relevant to implementing a system to assist programmers understanding program code. This section describes these findings.
In addition to making annotations on the code, nine participants made notes using separate pieces of paper. Three of these participants used two pieces of paper while the rest used a single piece. Only one of the participants added notes to both sides of the piece of paper. This resulted in a total of thirteen pages of notes; four were text only. The remaining pages contained diagrams and text (see Figure 3.6).

The most common usage for notes is class diagrams and flow diagrams. In class diagrams, the participant is drawing a graphical representation of the classes. There were six class diagrams drawn. A flow diagram shows a flow of either information or control through the code. There were five flow diagrams drawn.

The class diagrams typically included the interfaces and their relationships to classes. These were mainly drawn at an overview level. Only one of the diagrams contained property and method members. In addition, these members were a subset of the full members for the class. One of the diagrams was text only; the rest contained text and graphics elements. These ranged from simple
arrows connecting classes and/or interfaces to boxes similar to UML format. Two of the diagrams contained textual notes in addition to the diagram (see Figure 3.6). These were notes explaining the purpose of some of the classes.

The flow diagrams were more varied. Three diagrams worked at a high level (i.e. server and client) and showed some of the flows between these components. One of these had the relevant classes written for each component, another had the methods and the third was a conceptual diagram. The other flow diagrams were all based at the class level. They had some of the classes in the code and the messages that were passed between them. Two flow diagrams had textual notes. One set of notes listed the data structure being passed between components; the other listed some background information to the data flows.

Two pages contained notes of findings (see Figure 3.6). These findings were information the participant thought relevant to share with the junior programmer. Both of these pages were text only. They were both grouped lists. The first line
in each group was the heading and the remaining lines were indented.

The pages were matched with two of the basic reasons: working information and information for sharing. During the interviews, the participants stated the information was either for themselves or for sharing with the junior programmer. Three of the participants started writing notes with the intention of sharing. The other six participants intended the notes to be only for themselves; these notes were working information. Three of these participants stated that they would share their notes with someone else; however, they stated they would not directly share the working notes as they did not consider the information complete. They expected to either sit down with the person reading them or to revise them into a form that could be shared.

The time when the participants wrote the notes also varied widely. Two participants started writing notes very soon in the reading process. These participants did not make any annotations on the code; instead, they used the separate pages for writing information. Five of the participants started writing notes later in the process. These participants used a combination of annotations on the code and separate note pages. The remaining two participants annotated the code as they read. Near the end of their reading, they reviewed their annotations and wrote notes on separate paper.

The reason for the notes was linked to when the participant started writing them. The two participants who wrote the notes at the end of the session wrote the notes specifically for the junior programmer. In contrast, the two who started writing at the start of the session wrote the notes for their own understanding. The remaining participants were split between writing for the other person and for themselves. These were also the participants who changed the intention of the notes.

**Spreading Out and Organising Code**

One of the first things 11 of the participants in the study did was split the code into separate files (most files contained a single class, one contained two). They were originally given a single stack of paper containing all the code and they took this stack and leafed through to find how many different files there were. Then they separated the files into separate stacks of paper.

All the participants spread these stacks of paper around the desk (see Figure 3.7). They reported a variety of reasons why they did this: to keep the files separate,
to gain an overview of the code, to know where the various files are (for moving between them). Some of the participants combined this separating of pages with annotations. For example, P02 wrote the class names on the top of the first page in each class so he could quickly find them again.

Seven of the participants combined the separation of pages with spatial positioning. Once they had separated the piles they tended to keep them in the same positions. They would pick up a pile to read or search through it and then return it to the same position. For four of these participants, this was a conscious behaviour: they had deliberately made the decision to do this when separating the piles. One participant (P11) had even arranged the location of the piles based on their sizes. The underlying rationale was the bigger piles were more likely to be used so he placed them in locations where he could quickly reach them. The other three participants used spatial positioning without having made a conscious decision about it. One participant, P05, stated that he did not originally intend to keep them in the same locations but midway through the read task found it was easier to find things if he returned them to the same location.

3.3.6 Research Questions

These results provide answers to the four questions in this study:

1(a) Do programmers annotate on program code?
1(b) What are the types of annotations added?
1(c) How do these types differ from previously reported studies?
1(d) What are the main reasons for adding any annotations?

For 1(a), the answer is experienced programmers do annotate program code on paper. Thus, they might use digital ink to annotate code in an Integrated Development Environment if it were possible. Therefore, it is worthwhile developing a tool to allow freeform annotations in an Integrated Development Environment.

For 1(b), the annotations can be classified into eleven types. While each type has slightly different characteristics, they can be grouped into five categories: single line, multiple line, connector, complex and attentional. These types helped guide the development of vsInk; especially in regards to the recognition and refitting of annotations (see Chapters 4 and 5).
For **1(c)**, the reported types were similar to the annotation types reported in other studies. For example, these types are all included in both Marshall (1998) and Wang and Raghupathy (2007). Given that the reported types are similar, it may be possible to apply previously investigated functionality for annotations on program code. One example of where previous results are used in this thesis is the refitting study (see Chapter 5).

For **1(d)**, there were three main reasons: navigation, working information and information sharing. In addition, there were two less frequent categories of attentional and unknown. I speculate these reasons may be useful in developing a freeform annotation tool for programmers in the future. These reasons may also help determine any overlap between already existing tools and what is lacking, although this analysis was not performed.

### 3.4 Discussion

There were two hypotheses posited for this study:

**H1:** Programmers will still annotate the code in a similar way to annotating other forms of text.

**H2:** The main reasons for annotating would be to reduce the reader’s mental workload (computational offloading). Problem reformulation together with temporal and spatial constraining would be the main approaches used.

The results provide some evidence confirming both hypotheses, although not in their entirety.

For **H1**, all but one participant added annotations or wrote notes. In addition, the frequency of some annotation types was similar to what has been reported in other studies (Marshall and Brush, 2004). Underlines and highlights had similar frequencies to what has been reported before. This may be because they are so simple and easy to add.

However, there are some subtle differences to previously reported results. One annotation type that is more frequent in this study is compound annotations. Most of the compound annotations consist of a connector plus another type of annotation. I posit this is because annotations need to be associated with specific lines of code. If the participant did not need a precise anchor they would not use
a connector but would use a sidebar as is seen in prose annotation. This contrasts
with Marshall and Brush (2002), who found that compound annotations were
least common for personal annotations (although they were more common for
public annotations). This may be due to the nature of the task: the participants
may be attempting to build a closer link to the associated code segment.

The results partially support **H2**. The annotations were primarily added for
two reasons: navigation and recording information. While these are both ways
of reducing mental workload, I had posited that problem reformulation and
temporal and spatial constraining would be the main reasons.

Most of the annotations were added to help the participant navigate the code,
which is a form of temporal and spatial constraining. This is similar to what
other research has shown for when programmers read code (Sillito et al., 2005,
Singer et al., 2010, Maalej et al., 2014). The participants used a range of strategies
to mark specific sections of code (such as highlights and margin bars) and
also cross-reference marks to indicate connections between code on different
pages. For example, most participants highlighted some method names. This
selective highlighting, while similar to Integrated Development Environment
colour syntax highlighting, is more specific in that the participants highlighted
only some method names.

Another common form of navigation annotations was for backtracking. Again,
this is functionality that Integrated Development Environments already provide.
But Integrated Development Environment navigation support is generic where
annotations are specific. The readers annotated to focus on what they are inter-
ested in and to reduce the workload by place marking. This type of annotation is
seen in other studies (Johnson and Nadas, 2009, Crisp and Johnson, 2007) but the
importance of navigation in code comprehension makes it more critical for code
understanding. Digital ink annotation in Integrated Development Environments
could be a valuable aid to support code navigation.

Computers can, and do, generate a lot of navigation information but they do
not know what the reader is interested in. Therefore, the reader may take false
routes to find what they need, sometimes even backtracking to revisit previously
scanned code (Maalej et al., 2014). While annotations cannot help find unvisited
code they can help reduce the clutter of backwards navigation. A combination
of computer generated navigation links and user ink annotation may be optimal.

The other main reason was for recording information. Offloading information to
paper reduces the amount of attentive memory the reader needs while reading (Johnson and Nadas, 2009, Parnin and Rugaber, 2012). P08 mentioned that one of his challenges was trying to remember everything (“it felt like I was juggling several plates at once”). This reduction in memory requirements that freeform annotations afford is consistent with other studies (O’Hara and Sellen, 1997, Morris et al., 2007). This suggests that freeform ink annotations inside code editors could offer similar benefits.

One reason for adding annotations that was not observed was reformulating the problem. Examples of this would be drawing diagrams or recording changes in variables over the code. None of the participants added any annotations like these. However, they did add notes to similar effect on separate pieces of paper. This indicates the underlying mental processes may be similar but using a different mechanism than proposed.

This points to a difference in how annotations and separate notes are perceived and used. Annotations on the code were shorter and more cryptic, while the notes were longer and more explicit. This may be because the notes are longer-term artefacts that the participants may want to keep. These results corroborate the ideas behind CodeGraffiti (Lichtschlag et al., 2014). It may also be because of the contextual information for annotations (Jackel, 2014). The reader may be relying on checking the proximity of the annotation to retrieve any relevant information.

The spreading out of the paper on the table is an interesting observation that could benefit from further research. In particular, what is the importance of spatial positioning? There are tools that allow graphical representations for code (Bragdon et al., 2010, DeLine et al., 2012). These tools should maintain the spatial positioning of the elements to help programmers remember where things are. Large screen or multiple screen systems may also be beneficial by providing a larger space for programmers to work. The programmer could move the code files into positions that they find useful. The current file would be moved into a prominent location for work, then, when finished, it would be returned to its previous location. If the programmer needs it later they can easily find where it is based on its location. There is no research about this type of programmer support.

---

2This is the focus of my next research project (Sutherland et al., 2016c).
In code editing tools the main way to record information is by adding comments in the code. Previous studies comparing textual annotations vs. freeform annotations found people prefer freeform annotations (O’Hara and Sellen, 1997, Morris et al., 2007). O’Hara and Sellen (1997) suggested the most likely reason for this is freeform annotations do not interrupt what the reader is thinking about, therefore freeform annotations may be useful for storing information against the code without reducing the reader’s capacity for comprehending what they are reading. Second, annotations are not limited to text. Several of the participants drew diagrams of how they understood things. Studies have found that diagrammatic annotations provide value by keeping the diagram close to its context (Jackel, 2014). Finally, annotations stand out from the text. Freeform annotations look very different from the underlying document. Many of the participants in the study were able to quickly find things by just scanning through the code and looking for the annotations. This, when combined with spatial memory, allowed them to easily find something they had previously written.

3.5 Limitations

First, participants were given a specific task to do (read the program code for understanding). A different task may have resulted in different annotation patterns (e.g. marking an assessment, adding unit tests). Second, most of the participants used Integrated Development Environments that provide a variety of tools to assist with understanding. Some of these tools might be replicable on paper (e.g. an index of all the method locations). If these tools were available, the participants may have used different types of annotations. However, I contend that the selective nature of annotations may, in some circumstances, be better than generic functionality. Third, the programming language may have an effect on the types of annotation. Some languages are more difficult to read: a programmer reading program code in these languages may add different types of annotations. Finally, this study only had 13 participants. Including more participants might make some of the patterns observed more obvious. However, this study does provide some insight into how annotations could be useful in an Integrated Development Environment.
3.6 Summary

This study investigated how and why experienced programmers annotated code on paper. Previous work has shown it is possible to combine freeform ink annotations within a code editor (Sutherland and Plimmer, 2013, Lichtschlag et al., 2014) but there was no evidence of if, how and why programmers would annotate code with digital ink.

The results of this study indicate that somewhere to record information would assist programmers to comprehend what they read. The ability to annotate on code with freeform ink may be useful if the functionality was available. Separate blank pages may also be beneficial; especially if there is some way to link the notes to the code (Lichtschlag et al., 2014).

This study shows that programmers use annotations to assist with navigation, record information as working notes and to remember information for sharing. Using annotations for navigation confirms the results from the literature review in Chapter 2. Recording information for working notes and sharing are additional avenues of investigation. While some of this functionality is currently available in code editing tools, user specified particular navigation paths, which were frequently used in the study, are not supported in Integrated Development Environments. Furthermore, freeform ink annotations provide an alternate avenue that reduces the mental work needed when reading code. This is because ink annotations are quicker and easier to add compared to text-based annotations, allow greater expression and stand out from the code because they are visually distinct from the text.

Thus, this chapter shows that freeform digital ink annotations may be valuable for assisting comprehension. The next challenge is how to investigate freeform ink annotations in an Integrated Development Environment, as this study only used paper, which lacks the rich toolset that Integrated Development Environments have. However, there are still many technical challenges to adding annotations in an Integrated Development Environment.

The next chapter explores one of these challenges: how to accurately classify the annotation type. As the results show, annotations can be classified into five categories with different characteristics. Once annotations are classified into these categories it becomes possible to adapt them.
User Study 2
Return of the User

“What is research but a blind date with knowledge?”

Will Harvey

The previous chapter describes a user study investigating how and why programmers annotate. This study revealed the types of annotation added by programmers are similar to other reported types but with some subtle differences. One important difference is compound annotations are more common. These typically consist of a connector and an associated annotation. Therefore, vslnk will need to correctly recognise connector annotations in order to refit them. Automatic recognition in an Integrated Development Environment is not covered in the current research (see Chapter 2). This chapter describes my investigations into how recognition could be improved.

Section 4.1 outlines the context for classifying annotation strokes. Section 4.2 provides an overview of how the study was performed. The following two sections (§4.3 and §4.4) describe each phase of the investigation. Each section describes the implementation, explains the methodology used to evaluate the implementation and the key results. Section 4.5 answers the research questions for this study and discusses other areas of relevance. Finally, section 4.6 summarises the study.
The study described in this chapter was conducted with the approval of the University of Auckland Human Participants Ethics Committee. A copy of the approval letter is in Appendix A.

4.1 Introduction

Accurate recognition is important as it is a precursor to other operations on annotations (see Chapter 2). To group, anchor and adapt annotations the system must know the type of annotation, as different techniques are needed for each type. Errors in these operations can result in the annotation losing its meaning; for example, connecting the wrong points. It is also challenging to implement.

Studies on state-of-the-art recognizers typically report high accuracy rates (e.g. Chang et al., 2010, Field et al., 2010, Plimmer et al., 2012). However, these accuracy rates are specific to the implementation and dataset used (Field et al., 2010).

Each annotation is unique, with a special meaning to the creator. However, annotations normally fall into particular types (classes) based on their visual characteristics and semantic meaning (Marshall, 1997). Some of the types are visually very similar but semantically different; therefore it is difficult to classify annotations with different meanings but similar visual characteristics. As the results show, accurate automatic recognition is beyond the capability of current recognizers – particularly with regard to underlines and connectors.

One limitation of sketch-based recognizers is they do not include contextual information from the underlying document, such as the words, lines and paragraphs of text. Thus one approach for improving recognition would be including document information. Previous studies suggest that contextual information is relevant for separating certain classes of annotation (Rodríguez et al., 2008, Agrawal et al., 2010). However, most recognizers reported in the literature only use temporal features and/or features derived from the spatial relationships to other strokes.

An alternative approach, to ensure high accuracy, is to force the user to classify their annotations manually (Bargeron and Moscovich, 2003). This can result in 100% accuracy but at a cost of increasing the user’s workload. For tasks that require the user’s focus, they will be distracted by the manual classification, resulting in a less than desirable work environment. While the study by Bargeron and Moscovich (2003) was able to achieve 100% accuracy results, this was in
Chapter 4. User Study 2

a controlled test situation. During the annotating task, the participant was required to correctly classify their annotation as they added them\(^1\). It is unknown whether readers would maintain the same level of accuracy in a natural reading task.

To overcome the weaknesses of both automatic and manual classification, several alternative approaches were explored in two phases. In the first phase of the investigation, including contextual features from the document as input for the recognizer was explored. In the second, collaborative intelligence was implemented (Jones et al., 1998); this involves the user and the automatic recognizer working together to classify annotations.

This study investigated the following hypotheses:

**H3**: Contextual information about the location of the annotations will improve the accuracy of recognition.

**H4**: Involving the user in the recognition process will improve the accuracy of recognition.

In addition, the following sub-questions were asked:

2.1(a) How does adding contextual information influence the accuracy of recognition?

2.1(b) How does adding user constraints influence the accuracy of recognition?

2.1(c) How does allowing the user to change the results of classification influence the overall accuracy of results?

The next section describes how the research questions were explored.

### 4.2 Study Approach

The questions were investigated in two phases. During the first phase, the focus was on automatic recognition. The results from the first phase were used as the foundation for collaborative recognition in the second phase.

In the first phase, three different recognizers were implemented. The first two used RATA and the third one used a rule-based engine:

1. Standard RATA (baseline) (Blagojevic et al., 2011).

\(^1\)The participants were required to correctly classify their annotations as the focus of the study was refitting. The authors postulated that incorrectly classified annotations could confound the results of the refitting.
2. Standard RATA, plus additional context-specific features (extended RATA).

All approaches used a one-off approach for classifying annotation types\(^2\). Previously, only recurrent approaches have been reported for recognising freeform annotations. However, the one-off stroke approach has been used in the sketching domain with high accuracy rates (Blagojevic et al., 2011). This was identified as an area for potential research.

The three approaches were evaluated using datasets collected in vsInk. Freeform ink annotations were collected from participants, labelled and stored together with the document. The recognizers were then used to classify each annotation in an offline mode\(^3\). As the two RATA recognizers were separate components, ten-fold cross-validation was used for detecting differences.

Based on the results from phase one, two of the recognizers were used in the second phase. This phase evaluated whether including the user in the recognition process would improve the accuracy compared to automatic recognition only. The three conditions for this phase were:

1. **Automatic**: Automatic recognition only.
2. **Constraint**: Automatic recognition with a constraint that biases the recognition result based on where the annotation was added.
3. **Verification**: Automatic recognition; the user is shown the result and allowed to change it.

Freeform ink with constraints has been applied in the sketching domain to simplify the recognition process. For example, Zeleznik et al. (2008) use constraints to automatically calculate mathematical equations and both Eggli et al. (1997) and Michalik et al. (2002) use constraints to calculate B-splines and surfaces. My approach used a single constraint based on where the user was adding digital ink to the screen.

The **verification** approach is based on the work of Bargeron and Moscovich (2003). In this work, they used post-entry user input to select the annotation type. While they did not report accuracy rates, the rationale for this approach was 100% accuracy of recognition. Rather than forcing users to classify every stroke, vsInk uses RATA.Gesture to perform the initial classification. This result is displayed

\(^2\)A one-off approach is when each stroke is processed only once. In contrast, a recurrent approach re-processes every stroke when a new stroke is added. See §2.3.1 for additional details.

\(^3\)Offline here refers to using the recognizers outside of vsInk.
to the user in a manner that allows them to change the result if desired. This allows the combination of automatic recognition and user validation.

The next two sections describe each of the phases in detail, including what was implemented, how the functionality was evaluated and the results.

### 4.3 Phase One

The first phase evaluated the effectiveness of automatic recognition. To support the evaluation, vsInk was enhanced to recognise five of the main classes of annotation identified in Chapter 2. Figure 4.1 shows examples of these five classes. These classes are:

- Horizontal lines;
- Vertical lines;
- Enclosures;
- Connectors;
- Other (attentional, arrows, text and drawings).

The remainder of this section describes the implementation details, followed by how the recognizers were evaluated and the results of the evaluation.
4.3.1 Implementation

The three automatic recognizers were developed as plugins to vsInk. These plugged into the pipeline for adding annotations (see Appendix B). To allow for switching between the recognizers a configuration switch was added to vsInk. In addition, the two RATA recognizers can be executed independently of vsInk: this allowed for ten-fold cross validation. The following describes how each recognizer was implemented.

Baseline RATA Recogniser

RATA.Gesture (Chang et al., 2012, Plimmer et al., 2012) provides the baseline implementation. Previous studies reported accuracy rates from 94.4% to 97.6% for diagrams (Chang et al., 2012). vsInk includes RATA.Gesture in a pass-through module due to differences in the ink formats between the frameworks used for RATA.Gesture and vsInk. The pass-through module would convert the ink format used by vsInk into the format needed by RATA.Gesture. RATA.Gesture would then process the stroke and return the result.

The baseline recognizer classified horizontal lines (highlights and underlines), vertical lines (braces and margin bars), enclosures (circles), connectors and text.

As RATA.Gesture uses a model built by machine learning, an annotation-specific model was built for vsInk. This model used data collected in an offline mode from several volunteers. Each volunteer added between 10 and 15 strokes of the five annotation types. Once the model was generated it was included in the source code for vsInk and not changed.

The baseline recogniser uses the default recogniser from RATA.Gesture (Chang et al., 2012). The default recogniser in RATA.Gesture uses an ensemble of four machine learning algorithms (Bayesian Network, LogitBoost, Logistic Model Trees and Random Forest) combined using voting to train the model. The RATA.Gesture toolset provides a semi-automated, codeless process for generating models (Plimmer et al., 2012).
Table 4.1: Additional Contextual Features for Classifying Strokes in RATA.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF1</td>
<td>Stroke Line Height</td>
<td>The height of the stroke in lines of code.</td>
</tr>
<tr>
<td>CF2</td>
<td>Stroke Character Width</td>
<td>The width of the stroke in characters of code.</td>
</tr>
<tr>
<td>CF3</td>
<td>Context Overlap</td>
<td>Measures the overlap between the stroke and the underlying context.</td>
</tr>
<tr>
<td>CF4</td>
<td>Pen Type</td>
<td>Uses the type of the pen.</td>
</tr>
<tr>
<td>CF5</td>
<td>Start Point Location</td>
<td>Where is the start point of the stroke relative to the underlying context.</td>
</tr>
<tr>
<td>CF6</td>
<td>End Point Location</td>
<td>Where is the final point of the stroke relative to the underlying context.</td>
</tr>
</tbody>
</table>

Extended RATA Recogniser

The extended RATA recognizer was also implemented using a pass-through module. The module converted the ink then used the feature calculation functionality in RATA.Gesture to calculate the standard feature set. The module then calculated and appended several additional features to the set (see Table 4.1) and passed the complete feature set to the RATA.Gesture classification functionality. This functionality then returned the result. The model for these features was generating using the same offline dataset as the baseline model.

The following describes the new contextual features for the extended recogniser.

Stroke line height \((CF_1)\) and stroke character width \((CF_2)\) are contextual versions of stroke height and stroke width respectively. Stroke line height is calculated by dividing the height of the stroke’s bounding box by the line height\(^4\). This is equivalent to the height of the annotation in lines. Stroke character width is likewise calculated by dividing the width of the stroke’s bounding box by the character width\(^5\). This is equivalent to the width of the stroke in characters. Thus, these features extend the baseline RATA features with contextual information about the line height and character width.

Context overlap \((CF_3)\) is a measure of how much the stroke overlaps the underlying code (see Figure 4.2). This is calculated by first calculating the total area for the stroke (this is calculated as the area of the bounding box - \(B_{\text{Stroke}}\)). Second, the area for the strokes over the top of the code is calculated (\(B_{\text{Intersect}}\)). The final ratio is calculated by \(\text{Ratio} = \frac{B_{\text{Intersect}}}{B_{\text{Stroke}}}\). This is used as one measure for

\(^4\)For simplicity the line height is assumed to be a constant value.
\(^5\)As Visual Studio uses a fixed width font, this value is always constant.
distinguishing underlines from connectors. The closer this ratio is to one, the more likely a line is to be an underline. Thus, this will be zero when all the stroke is outside the context and one when the stroke is entirely within the context.

*Pen Type* ($CF_4$) differentiates between underlines and highlights. This is zero for a normal pen and one for a highlighter.

*Start Point Location* ($CF_5$) and *End Point Location* ($CF_6$) both record where the stroke is relative to the code: Start Point refers to when the pen first connects with the screen; End Point refers to the last point prior to disconnecting. These values are -1 for the left of the context, 0 for within context and 1 for the right of context.

**Heuristic Rules Recogniser**

The heuristic rules recognizer is based on the research by Golovchinsky and Denoue (2002) and Stahovich and Lin (2016). These studies showed that application-specific features can produce comparable results to using a full feature set.

The recognizer contained rules for classifying five classes of strokes (see Table 4.3). These rules used features derived from RATA plus some additional context-specific features. These features are listed in Table 4.2 and detailed below.

These features can be thought of as functions (see Figure 4.3). The functions are summed together to provide a probability for each stroke class (see Table 4.3). The class with the highest probability is then chosen as the final class.
Table 4.2: Heuristic Features for Classifying Strokes. Features with a ‡ are derived from RATA.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Straightness (S) ‡</td>
<td>Measures the straightness of the stroke.</td>
</tr>
<tr>
<td>Curviness (C) ‡</td>
<td>Measures the amount of curvature of the stroke.</td>
</tr>
<tr>
<td>Height to Width (HW) ‡</td>
<td>Measures the ratio of the height of the stroke to its width.</td>
</tr>
<tr>
<td>Context Overlap (O)</td>
<td>Measures the overlap between the stroke and the underlying context.</td>
</tr>
<tr>
<td>Note Boundary (B)</td>
<td>Measures how many endpoints are within the note area of the document.</td>
</tr>
</tbody>
</table>

Straightness is calculated as the ratio of the sum of the lengths between the points to the Euclidean distance between the start and end points. This feature uses the same formula as feature $X_{SR}$ from Stahovich and Lin (2016). This is used to measure how close the stroke is to a straight line: the closer the ratio is to one, the more likely the line is to be straight. This is used to differentiate lines from non-lines (e.g. circles and text).

Curviness is calculated as a measure of how curvy a stroke is. This is calculated by...
summing the absolute angle for each point. This feature uses the same formula as feature 1.11 from Blagojevic et al. (2010). Features with a low total angle are more likely to be lines; a total angle close to 360 indicates a circle. As the curviness exceeds 360 the stroke is most likely to be text.

*Height to Width* is calculated as one of two ratios. This feature is based on feature 7.19 from Blagojevic et al. (2010). When the height is greater than the width then it is calculated as \( \text{Ratio} = 1 - \frac{\text{Width}}{\text{Height}} \); otherwise it is calculated as \( \text{Ratio} = \frac{\text{Height}}{\text{Width}} - 1 \). If the height and width are the same, this will result in a ratio of zero. When the height is greater than the width, the ratio approaches closer to one as the height becomes greater than the width (\( \lim_{\text{Height} \to 0} \text{Ratio} = 1 \)). When the width is greater than the height, the ratio approaches negative one as the width becomes greater than the height (\( \lim_{\text{Width} \to 0} \text{Ratio} = -1 \)). This feature differentiates between vertical and horizontal lines.

*Context Overlap* is a measure of how much the stroke overlaps the underlying code. This is calculated using the same formula as \( CF_3 \). This is used as a measure of distinguishing underlines from connectors. The closer the ratio is to one, the more likely it is to be an underline.

The final feature, *Note Boundary*, is based on how many endpoints are within the highlighted areas. The value of this feature is the number of endpoints in the highlighted area (zero, one or two) multiplied by fifty. If there are no endpoints then no bias is applied. If there is one endpoint, then the result is biased towards being a connector. If there are two endpoints, then the result is biased toward being text.

### 4.3.2 Methodology

Two different approaches were used for comparing the accuracy of the classifiers.
For the two RATA classifiers, a single dataset of annotations was collected from a group of volunteers. Each volunteer drew between 10 and 15 annotations of the five main classes (horizontal lines, vertical lines, enclosures, connectors and other). As the participants drew the annotations for each class separately, the annotations were automatically labelled as the strokes were collected. During the collection phase, the features for the strokes were calculated as each stroke was entered: both the standard and extended feature sets were calculated for each stroke. At the end of the collection session all the strokes, together with their labels and features, were saved. There was one saved file per participant.

To detect any differences between the approaches ten-fold cross validation was used. The dataset was split into ten segments and evaluated ten times. For each evaluation, models were generated using nine of the segments. The tenth segment was used to test the accuracy of the models.

The testing phase produced a data stream with three pieces of data. The first was the label of the annotation (as set during the collection), the result of the baseline recognizer and the result of the extended recognizer. A post-testing step then marked the results of both recognizers: if the result matched the label it was scored one, otherwise, it was scored zero. If the classification result was different between the two classifiers, the stroke and the two results were stored in a log file.

The full process (data segmentation, model generation and model testing) was completely automated and executed using a virtual machine in the high-performance machine cluster. This process produced a report listing the main differences between the two feature sets and the comparative accuracy from each fold.

In contrast, comparing baseline RATA against the heuristic rules involved a second set of data. This data was collected later, as the heuristic rules classifier was implemented after the RATA classifiers were evaluated. As the participants added annotations to vsInk vsInk would classify and record the results from both the baseline and heuristic rule classifiers. Then during the interview stage the participants identified any incorrect classifications from the baseline and provided the correct classification. In addition to logging the result of the classification, the time required for each classification was also recorded.

During the analysis phase, these three data streams were collated and used to produce a single table listing the three classifications. Again, each result
was marked and scored against the annotation label (the classification from the participant) and a list of differences produced.

Both sets of results were analysed using R 3.2.2 (R Core Team, 2015). Inferential tests (Student’s T-Test and ANOVA) were used in the analysis.

### 4.3.3 Results

To evaluate the two RATA recognizers, 636 annotations were collected. For comparing RATA and the heuristic rules, 393 annotations were collected. Table 4.4 shows the breakdown of annotations collected for each category.

An annotation was defined as correctly classified by a recognizer when the result matched the label of the annotation (either from the data collection or as specified by the participant).

There were no differences in the classification results between either of the RATA recognizers. Each ten-fold produced exactly the same results for both recognizers (by both the summary and the list of difference reports.) This implies that the extra features did not change the classification at all.

When comparing the baseline to the heuristic rules, the T-test did not show any evidence of a significant difference in overall accuracy between the two classifiers ($t(397) < 0.01$, $p = 0.99$). However, a two-way ANOVA did find a significant interaction between the classifier and the annotation class ($F(2, 790) = 9.24$, $p < 0.001$). Looking at the list of differences (see Table 4.5), it appears the heuristic recognizer was better at differentiating horizontal lines from connectors while RATA was better for differentiating enclosures from horizontal lines. There is no evidence of any difference between the recognizers for classifying vertical lines ($t(28) = 1.20$, $p = 0.12$).

**Table 4.4: Number of annotations collected for evaluating the classifiers.**

<table>
<thead>
<tr>
<th>Class</th>
<th>RATA</th>
<th>Heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal Line</td>
<td>130</td>
<td>69</td>
</tr>
<tr>
<td>Vertical Line</td>
<td>126</td>
<td>60</td>
</tr>
<tr>
<td>Enclosure</td>
<td>133</td>
<td>63</td>
</tr>
<tr>
<td>Connector</td>
<td>124</td>
<td>100</td>
</tr>
<tr>
<td>Other</td>
<td>123</td>
<td>101</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>636</td>
<td>393</td>
</tr>
</tbody>
</table>
TABLE 4.5: Accuracy of each recognizer by annotation class. The first number is the number of correct matches; the number in brackets is the percentage of the total.

<table>
<thead>
<tr>
<th>Class</th>
<th>RATA</th>
<th>Heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal Line</td>
<td>42 (60.9%)</td>
<td>63 (91.3%)</td>
</tr>
<tr>
<td>Vertical Line</td>
<td>21 (35.0%)</td>
<td>26 (43.3%)</td>
</tr>
<tr>
<td>Enclosure</td>
<td>57 (90.5%)</td>
<td>28 (44.4%)</td>
</tr>
<tr>
<td>Connector</td>
<td>63 (63.0%)</td>
<td>59 (59.0%)</td>
</tr>
<tr>
<td>Other</td>
<td>16 (15.8%)</td>
<td>27 (26.7%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>199 (50.6%)</td>
<td>203 (51.7%)</td>
</tr>
</tbody>
</table>

There is no evidence of any difference in classification rates between the recognizers for type ‘other’ ($t(28) = 0.46, p = 0.65$); although inspection of the results showed a potential bias for each recognizer. RATA tended to classify other annotations as either horizontal or vertical lines. The heuristic rules tended to classify them as enclosures.

In addition, there is strong evidence of a difference between the speed of the recognizers ($t(397) = 22.15, p < 0.001$) 95% CI of 25ms to 31ms. However, while the difference is statistically significant, in terms of human perception this is not noticeable. A second speed difference is the one-off load needed for RATA. This would typically take between 15 to 30 seconds; although this does depend on the machine specification.

### 4.4 Phase Two

The findings from the first phase are rather disappointing. Due to the first phase being an exploratory investigation, a decision was made to change the focus of the study from a pure automatic approach to a collaborative one. This allowed investigating how changing the user interface impacted recognition.

#### 4.4.1 Implementation

Three approaches were implemented for this phase. These all plug into the pipeline architecture of the adding operations (see Appendix B). The following sections now describe how these approaches were implemented.
Figure 4.4: Screenshot of the verification approach for classifying annotations. The top image shows the display when the annotation is added; the bottom image shows the pop-up with the list of stroke types.

**Automatic Only**

In the automatic only mode, the baseline RATA recognizer is used. Once the stroke is classified the user is unable to change the classification.

**Verification**

This approach uses the baseline RATA classifier to perform the initial classification of the annotation when the stroke is added. It then displays a pop-up showing the classification result (see Figure 4.4). The user can then click on the pop-up to change the annotation type. However, this is not required as if the user is satisfied with the automatic classification they can leave it.

The pop-up is automatically hidden when a new stroke is added or the screen is scrolled.
Constraint

This approach also uses the baseline RATA classifier. However, a visualisation, the blue section, is added to the screen dividing it into two areas (see Figure 4.5). If the stroke is only on the left side it will be classified using the RATA recognizer. If the stroke is only on the right side it will be classified as a text annotation. Finally, if the stroke crosses from one side to the other it is classified as a connector. The user is unable to change the classification type once the stroke has been added.

4.4.2 Additional Functionality

In addition to the recognizers, a post-it note visualisation was implemented for this phase. Two visually similar strokes with different meanings are connectors and horizontal lines. Horizontal lines are typically added as underlines while connectors are used to join a location within the text to another annotation. Whenever a connector is added in vsInk, a visualisation was displayed showing an associated note. This visualisation allows the user to see when the stroke was classified as a connector. This was implemented for all three conditions.

The note was drawn as a small yellow box to the right of the end-point of the connector stroke (see Figures 4.4 and 4.5). The end-point is defined as the right-most point of the stroke on the screen. The note is completely opaque and has a small drop-shadow to show it is separate from the underlying document. Adding strokes within the note caused the note to grow. Any strokes added
within a note were ignored by the classifier and grouped together as a single annotation.

4.4.3 Methodology

To evaluate the effectiveness of the machine collaboration approaches, a within-subject user study was performed. The study involved the participants performing a short reading for understanding. There were 17 participants in the study: they were a combination of graduate students and professional programmers. All participants were familiar with *Visual Studio 2013* and used it regularly. Eight of the participants had also used pen-based computers previously.

The protocol involved two questionnaires (pre- and post-study), the reading task and a post-study interview. At the start of the study the participants were welcomed and the purpose of the study explained. When they had given their consent, they were asked to fill in the pre-study questionnaire on their background. This provided some demographics about their programming experience and previous usage of pen-based interaction.

After the questionnaire, the task was explained and *vsInk* demonstrated. This included the three different classification approaches. The participant was then given five minutes to familiarise themselves with how *vsInk* worked.

For the actual reading task, the participants were given one of three code files to read. Each code file was over one thousand lines in length. This was both to ensure moderate complexity of the code and to ensure the participants scrolled the screen. The participants were given fifteen minutes to read the code. Every five minutes the annotation classification approach\(^6\) and code file was changed. The order of both classification approaches and files was randomly determined prior to the participant starting to read. During the reading task, *Morae* (Tech-Smith, 2015) was used to record the screen. During this time the participant was observed by the researcher to see what they were doing with the software and any issues they had.

After the reading task, the participant was given the post-study questionnaire about their experience. This questionnaire has three sections. The first section contains three statements with associated five-point Likert Scales (Strongly Disagree, Disagree, Neutral, Agree and Strongly Agree). These statements are:

---
\(^6\)To simplify the changing of classification approach during the reading task, a UI pane was developed during the implementation phase to quickly change the classification approach.
Chapter 4. User Study 2

- I enjoyed using the software while reading code.
- I found it easy to add annotations while reading code.
- I think that annotating code would help me understand the code.

The second section asks the participant’s opinion on each of the three approaches. For each approach, there are three statements using Likert Scales (using the same rating as above):

- I found it easy to select the class of annotation.
- I found the class of annotation was often correct.
- This is my preferred approach for selecting the annotation class.

The final section asks the participants to rank the three approaches; from most preferred to least preferred.

After the questionnaire, a semi-structured interview was carried out. Each annotation was reviewed with the participant and they were asked whether the classification was correct or not. If the classification was incorrect, the participant was asked what the classification should have been. They were also invited to share any issues or suggested improvements for vsInk.

After the study, the details from the questionnaires were recorded in an Excel spreadsheet. The annotation details were recorded in a second spreadsheet. For each annotation, the following details were recorded:

1. The participant number
2. The classification approach
3. The result of classification (RATA)
4. The actual annotation class (as specified by the participant)
5. Whether the participant changed the annotation classification or not
6. Whether the annotation was erased or not

The classification results came from the log files produced by vsInk. The actual annotation type came from the post-study interviews. The video recordings from Morae were used to identify which annotations were changed and which were erased. All the data were entered in an anonymous format so the participants could not be identified.

7To rate the accuracy of the classifier the Stroke Type adornment was used (see Table B.2 in Appendix B). During the data collection, this adornment was turned off. During the interview, the adornment was turned on allowing the participant to see the annotation type and rate the accuracy of the classifier.
TABLE 4.6: Categories of annotations added. The number in brackets is the percentage of Total Annotations Added.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal Line</td>
<td>139  (24.6%)</td>
</tr>
<tr>
<td>Vertical Lines</td>
<td>61   (10.8%)</td>
</tr>
<tr>
<td>Enclosures</td>
<td>87   (15.4%)</td>
</tr>
<tr>
<td>Connectors</td>
<td>237  (42.0%)</td>
</tr>
<tr>
<td>Other</td>
<td>40  (7.1%)</td>
</tr>
<tr>
<td>Total</td>
<td>564</td>
</tr>
</tbody>
</table>

The analysis was performed using R 3.2.2 (Bryer and Speerschneider, 2015, R Core Team, 2015). Likert charts and bar plots were used to visualise the results from the questions. Inferential statistical tests were used for analysing the accuracy and timing data. The tests used are listed in the results section below.

In addition, the observation and interview notes were transcribed. These were analysed using thematic analysis (Guest et al., 2012). Each set of notes was read and potential themes identified. After all the notes were read, these themes were grouped together where possible. The notes were then re-read and classified using these themes. This resulted in a set of classified notes and a taxonomy of themes.

4.4.4 Results

During the user study, 564 annotations were collected (see Table 4.6). There was no significant difference in the number of annotations collected for each approach ($F(2, 395) = 1.40, p = 0.25$).

The classes of annotations differ slightly from other reported counts. For example, Shilman and Wei (2004) and Wang et al. (2006) reported horizontal lines were approximately 40% and 42% of the annotations; while connectors were around 21% and 16% (Shilman and Wei, 2004, Wang et al., 2006). In contrast, the most common annotations were connectors (42.0%) with single line annotations comprising only 24.6% of the total annotation collected.
Table 4.7: Accuracy Rates for each Approach.

<table>
<thead>
<tr>
<th>Approach</th>
<th># Added</th>
<th>Mean % Correct</th>
<th>95% CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>218</td>
<td>49.1</td>
<td>42.4% – 55.7%</td>
</tr>
<tr>
<td>Verification</td>
<td>165</td>
<td>78.8</td>
<td>72.5% – 85.0%</td>
</tr>
<tr>
<td>Constraint</td>
<td>181</td>
<td>69.6</td>
<td>62.9% – 76.3%</td>
</tr>
</tbody>
</table>

Accuracy

An annotation was defined as correctly classified when the classification result matched the actual annotation type (as classified by the participant). Table 4.7 shows the average accuracy rates for the three approaches. The results were analysed using a one-way ANOVA with approach as the factor. As this showed a significant difference between the three approaches, $F(2,395) = 13.34$, $p < 0.001$, a Tukey’s HSD analysis was used to identify which approaches were different. There were significant differences in the accuracy rates between the baseline and constraint approaches ($p < 0.001$) and the baseline and verification approaches ($p < 0.001$). There were no significant differences between the constraint and verification approaches, ($p = 0.40$).

The two most commonly misclassified annotations were horizontal lines and connectors. A two-way ANOVA, with annotation classification and approach as the factors, identified a significant interaction between the two factors ($F(15, 739) = 2.91$, $p < 0.001$). The baseline approach had a bias towards connectors, with horizontal lines often being classified as connectors. The verification approach also had the same bias\(^8\), whilst the constraint approach had a bias towards horizontal lines. This may be because the participants intended to add a connector but did not cross over into the blue area.

One additional behaviour observed was erasing annotations: Table 4.8 shows the erasure rates. A one-way ANOVA was used with approach as the factor. This showed a significant difference between the three approaches ($F(2, 166) = 11.93$, $p < 0.001$). A Tukey’s HSD analysis identified the significantly different approaches. There is evidence of differences between baseline and verification approaches ($p < 0.001$) and constraint and baseline approaches ($p < 0.001$) but no evidence of any differences between the constraint and verification approaches ($p = 0.26$). The most commonly erased annotations were connectors followed by horizontal lines.

\(^8\)This is because it was using the same underlying classifier; after manual correction the bias was removed.
TABLE 4.8: Annotations Erased by Approach. The number in brackets is the percentage of \# Added.

<table>
<thead>
<tr>
<th>Approach</th>
<th># Added</th>
<th>Erased</th>
<th>Erased (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>218</td>
<td>75</td>
<td>(34.4%)</td>
</tr>
<tr>
<td>Constraint</td>
<td>165</td>
<td>37</td>
<td>(22.4%)</td>
</tr>
<tr>
<td>Verification</td>
<td>181</td>
<td>31</td>
<td>(17.1%)</td>
</tr>
<tr>
<td>Total</td>
<td>564</td>
<td>143</td>
<td>(25.4%)</td>
</tr>
</tbody>
</table>

FIGURE 4.6: Participants’ ratings about the general experience.

To identify why annotations were erased the participants were asked why they erased annotations during the interview. The most common reason stated was because the annotation was incorrectly classified. Other common reasons for erasure were because the annotation was a mistake (e.g. accidentally added) or the participant had changed their mind. Due to time constraints, it was not possible to review the erased annotations with the participants to find out the reason for each individual annotation.

User Feedback

Figure 4.6 shows the participants’ overall feedback on adding annotations to program code. In general, the participants liked the ability to add annotations. All the participants thought that annotations would help them understand the code. However, when asked, three of the participants stated they would not add annotations even though they thought they were useful. These participants thought that annotations would conflict with how they currently reviewed code. In addition, some participants did not see any value in keeping their annotations. They viewed annotations as temporary artefacts used during the review process that could be removed afterwards.

One of the key issues identified during the interviews was the accuracy of the inking. Many participants found it was difficult to draw where desired. Two
main factors were described as influencing the accuracy: the pen sliding on
the screen and parallax between the pen tip and the document. As these are
both hardware issues they are outside the scope of this thesis, see Agrawala and
Shilman (2005) for a discussion of these issues.

Figure 4.7 shows the participants’ responses for each approach and Table 4.9
shows their overall preference. All the participants agreed that collaborative
recognition is an improvement over the baseline approach. Overall, the verifi-
cation approach had the highest ratings for being easy to use, followed by the
constraint approach and then the baseline. Of interest, the perceived accuracy
swapped the two approaches: the constraint approach was perceived as being
slightly more accurate. Finally, the two collaborative approaches had higher
ratings for being the preferred approach.

During the interviews, there were some interesting points raised relating to these
results. First, most participants stated that none of the approaches was ideal:
each approach had some limitations or annoying issues.

The participants mentioned a number of issues with the verification approach.
The display in the verification approach was often perceived to be in the wrong
location and often it was not used. Some participants stated the display was
covered where they wanted to add the next stroke. Occasionally they would
click the display and be confused because a new stroke was not being added.
Another issue is participants would often add several strokes before realising
that an earlier stroke was incorrect. One participant stated they often forgot to
check the type was correct. One common suggestion was to add the ability to
change the type after successive strokes had been added; any strokes added after

**Figure 4.7:** Participants’ ratings for each classification approach.
TABLE 4.9: Participants’ preferences for classification approach. The number in brackets is the percentage of the total.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Total</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>3</td>
<td>(17.6%)</td>
</tr>
<tr>
<td>Constraint</td>
<td>7</td>
<td>(41.2%)</td>
</tr>
<tr>
<td>Verification</td>
<td>7</td>
<td>(41.2%)</td>
</tr>
<tr>
<td>Total</td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

the changed stroke could be retrospectively changed to text\textsuperscript{9}.

Issues mentioned for the constraint approach included wanting to add text in the code area and the blue highlight overlapping the code. The overlap issue prevented some participants from adding an underline when the code was covered by the highlight. While they realised it was possible to scroll the text, they often did not want to do this as it interrupted their flow.

Given these results, it is not surprising there was a mix of preferred approaches. These preferences are better understood in light of the participants’ comments. The participants who preferred the baseline approach liked the unconstrained nature of the application. All three participants did not regularly annotate on any books or other types of paper. Two of these participants saw annotations as temporary and only valuable during the reading session. As such, neither wanted to keep their annotations and thought that they should be deleted. Therefore, they felt there was no need to recognise the annotations’ type or refit them later. The third participant thought annotations would only be useful for drawing diagrams. As he only wanted one type of annotation he felt both approaches were overly constraining.

Notes

While not part of the usability study, the notes were frequently mentioned during the interviews. They were seen as both beneficial and annoying. While most of the participants liked them and thought they added value, some of the participants disliked them. These participants wanted some way to turn off the notes so it did “not interrupt their flow”.

Other comments included:

\textsuperscript{9}As these are suggestions from the participants during the evaluation, their feasibility has not been evaluated.
Chapter 4. User Study 2

1. Wanting some way to move the notes around;
2. Having a function to merge notes together;
3. Being able to hide either all or some of the notes;
4. Wanting the notes to automatically avoid text, annotations and other notes.

As notes are outside the scope of this thesis, these suggestions are avenues for future investigation.

Themes

From the interviews, three themes were identified. The first theme was difficulties in using the software. Examples of items in this theme include:

1. Recognition not completely accurate;
2. UI elements obstructing either the underlying code or other annotations;
3. Not automatically removing attentional marks;
4. Changing modes (e.g. the participant often forgot when they changed to erase mode);
5. Trying to use touch to scroll through the code;
6. The annotations should change when the underlying code is changed.

Some of these items are issues with the current implementation (1 and 2). Others are suggestions for future enhancements in the implementation (3 to 6). Some of these have been identified previously (e.g. attentional marks by Anderson et al. (2004b) and combining pen plus touch by Hinckley et al. (2010)).

The second theme is the potential benefits of annotations on code. Examples of items in this theme include:

1. The annotations standing out and being easy to find;
2. Adding was quick and easy;
3. Still possible to access the underlying functionality of the editor;
4. Adding annotations did not change the code.

These items show the ease of adding annotations. This allows the user to focus on the task (understanding the code) rather than the mechanics of annotating. At the same time, because annotations were different from the code it was easy to come back to them later. However, one associated issue was whether the annotations would still be easy to find in the future; especially if the code had changed.
The third theme was how the annotations could be used in the future. Examples included:

1. Collaboration;
2. Code review;
3. Teaching.

Most of the participants think that annotations are more useful in settings where other programmers are involved. Two of the participants even mentioned that including annotations into a code review tool would increase its value as they could both see the changes and annotate them.

### 4.5 Discussion

There were two hypotheses posited for this study:

**H3:** Contextual information about the location of the annotations will improve the accuracy of recognition.

**H4:** Involving the user in the recognition process will improve the accuracy of recognition.

There was no evidence that contextual information improved the accuracy of recognition. An unexpected result is both RATA classifiers produced exactly the same results. In contrast, I had proposed including contextual features would improve recognition.

The complete agreement may be due to the features chosen. The features implemented were very simple\(^{10}\) and may not have enough power to help distinguish similar classes of annotation. Studies using OneNote suggested a different set of contextual features and a reduced feature set overall (Wang and Raghupathy, 2007). Changing to the same feature set may improve recognition. Alternately, reducing the number of base features provided by RATA may increase the power of the contextual features that were implemented in vsInk. Finally, separating the strokes into categories prior to recognition may improve the classification. For example, Blagojevic et al. (2011) reported improved accuracy by separating text from non-text and Stahovich and Lin (2016) by separating cross-outs from non-cross-outs.

\(^{10}\)These were chosen based on discussions with other researchers.
Another reason for the agreement between recognizers may be due to noise in the data. The volunteers were asked to draw annotations as they would normally (one participant annotated in Japanese). This resulted in a large variety of annotations, even within each class. The contextual features might have insufficient power compared to the noise in the data. The studies could be re-run using a cleaner data set to determine whether the noise was a confounding factor. However, while this may improve the power of the contextual features, it is debatable whether it would improve the overall accuracy rates of the recognizers as real world data would still be messy.

There was some evidence supporting user involvement improved the accuracy of recognition. A one-way ANOVA between the baseline and collaborative approaches rates yielded significant variation in accuracy rates ($F(2, 764) = 23.41, p < 0.001$). A Tukey HSD analysis showed the variance was between the constraint and baseline conditions ($p < 0.001$) and between the verification and baseline conditions ($p < 0.001$). There was no evidence of a difference between the constraint and verification conditions ($p = 0.42$). However, none of the approaches had accuracy rates that matched previous studies. Previous studies for RATA reported accuracy rates in the high 90’s (Chang et al., 2012). Microsoft OneNote also reported accuracy rates in the 90’s (Wang et al., 2006, Wang and Raghupathy, 2007). In contrast, my maximum accuracy rate was only 85% (see Table 4.7). There are two possible reasons for this difference: the classification approach used and the participant’s intentions.

In this study, a one-off classification strategy was used. In this strategy, strokes are only classified once (when they are added.) There might not be sufficient information available to the classifier at this stage to correctly classify the strokes. In contrast, OneNote uses a recurrent classifier that re-classifies all the strokes when a new stroke is added. This allows the classifier to calculate features based on both past and future strokes (Wang and Raghupathy, 2007, Stahovich and Lin, 2016). This additional information may be sufficient to improve the overall accuracy rates. Reviewing the results from Wang and Raghupathy (2007) shows some similar trends. Both studies show annotations in the ‘other’ category (text, drawings, symbols, etc.) are difficult to classify and lines (horizontal or vertical) are easier. The main difference is connectors, with OneNote having a higher accuracy rate.

The second reason is user intentions. One assumption underlying this study is it is important to correctly identify all annotation types. Comments from some
of the participants suggest this may not be the case (e.g. “I do not care about that annotation, so I don’t need to set it as an underline.”) As mentioned in the introduction, accurate classification is a precursor for automatically adapting annotations. But some participants in this study indicated that their annotations did not need to be kept and therefore there is no need for automatic adaptation.

Adapting annotations is important if the annotations need to be kept for future reference and the underlying document can change. Some participants mentioned it would be useful to keep their annotations but only some of their annotations. This points to two different categories of annotation: transient vs. permanent annotations. In the implementation the focus was classifying all annotations: perhaps it would be better to involve the user in choosing which annotations to keep rather than classifying all types. This may also work better for the user as they could focus on those annotations they want to keep rather than being distracted by classifying all annotations.

Compared to previous work (Bargeron and Moscovich, 2003), the expected accuracy rates for the verification approach was lower than expected. As well as the user intention, this may be due to the implementation. During the task, people would add the connector and several strokes of text before realising the connector was not correct. As the implementation only allowed changing the last stroke they could not correct this. It may be better in future to allow classification of the first stroke in a set. But this raises the challenge of how to correctly group strokes together; a challenge just as difficult as classifying individual strokes.

The results also show the classifiers have different biases. Overall, the constraint approach was slightly better at classifying horizontal lines while the baseline approach was better at classifying connectors. This suggests that the approach used may vary depending on the task. The constraint approach would work better for tasks involving adding connected notes, especially when these notes are more permanent. The baseline approach would be better for unconstrained annotating where there is little or no need for automatic adaptation.

For the participants’ preferred approaches there was an even split between the constraint and verification approaches. This split may be because of different ways people approached the task. The participants who preferred the constraint approach are more likely to focus on reading and understanding: verifying the annotation class interrupts their thought processes and potentially distracts them from what they are reading (Bargeron and Moscovich, 2003). Thus the
constraint approach is more fluid to them. In contrast, those who preferred the verification approach fall into two groups. One group prefers accuracy over fluidity, while the other group prefers unconstrained annotations. The constraint approach allows some user input but the user is limited in both the input and what they can do. Thus, both are limited at the cost of increasing fluidity which may be an unacceptable cost to these participants. One suggestion during the interview was to include both approaches. This would allow the benefits of both approaches: the natural style while drawing and the ability to correct as required.

Another observation is the rate of erasure. The participants erased a quarter (25.4%) of the annotations they added. Participants erased annotations for two reasons: either they changed their mind or it was incorrect. People changing their mind is not a new phenomenon. This has been observed in other studies, both on paper (see Chapter 2) and electronically (Priest and Plimmer, 2006). The advantage of an electronic environment is the strokes can be completely removed, whereas on paper they remain and potentially obstruct other information. In this study participants would erase both entire annotations and strokes within an annotation. When asked, the participants did not mind having to delete multiple strokes. Some participants mentioned they preferred to have control over which strokes to delete rather than having the entire annotation deleted at once. For example, if someone is adding a note and misspelt a word they would only want to delete that word.

In contrast, incorrect classification is an artefact of the implementation. Previous studies often did not classify the annotation class\(^{11}\). Because the annotation was classified and a note area displayed, the participants were more aware of the annotation class. While most participants thought the note was useful, they all had issues with when the note was added. The baseline approach was most inaccurate: this approach also had the highest number of incorrect annotations erased. However, the other two approaches were still too inaccurate.

The most common reason for deleting a misclassified annotation was because either the note was added when not expected or vice versa. As mentioned in the implementation, the note is only added when a connector has been classified; therefore this issue is a direct result of misclassification. Even in the verification approach, participants would still delete misclassified annotations as they would

\(^{11}\)There is no need to classify the annotation class on static documents.
often add the text before realising the note had not appeared. This may be a training issue but as it interrupts the reader’s flow, it is still not an ideal solution.

An incidental feature of significance from this study is notes. Originally the notes were added to show a connector had been added. However, this mechanism suggests a solution for grouping text. Text is particularly problematic for adaptation as it consists of multiple strokes. Even the simple action of repositioning text can result in errors if the change splits the annotation (Marinai, 2013). Thus grouping text is very important.

Several participants liked notes as it showed them when strokes were treated as text. Thus, notes may be a natural solution that our participants are familiar with. They can be used without any additional need for training and they show the user exactly which strokes belong together. I posit that, as long as all the strokes on the note are moved together, text within a note will not have any adaptation issues.

However, the implementation of notes is still lacking. While most participants liked notes, several suggested improvements for them. One comment was the location of the note is not important (the anchor of the connector in the text is what is important). Thus notes with a connector could be moved around without losing their meaning. This presupposes that all the strokes in the note move together and that the connector refits to stay between the text anchor and the note. Second, participants do not always want to see the notes. Several suggested some mechanism for hiding (or collapsing) notes. Ideally, this would need two forms: an individual note or all notes. Third, notes sometimes overlapped. Therefore, some mechanism is needed to prevent this overlap. This could be hiding previous notes or having some form of reflow so notes avoid each other.

4.6 Summary

This chapter describes the investigations into how annotation recognition could be improved. While the results of this study are mixed, with some approaches working and others not, it provides some indications of what can be done. First, different approaches are better for different scenarios. This applies for both automatic recognition and collaboration. Second, while the focus was on improving accuracy for all annotations, the results suggest that focusing on identifying which annotations are important may produce better results.
Chapter 4. User Study 2

This study shows some potential approaches for improving classification. However, this is just a stepping stone: the reason for this study is accurate classification is needed prior to adapting annotations. The next chapter builds on the results in this thesis and explores how annotations can be automatically refitted when the underlying content changes.
User Study 3
The Curious Case of Annotation Adaptation

“Science is the acceptance of what works and the rejection of what does not. That needs more courage than we might think.”

Jacob Bronowski

The previous chapter described the investigations into improving the classification process. The aim was to provide more accurate recognition of the different stroke types. This is important because we cannot automatically refit annotations if we do not know what they are. Thus, Chapter 4 provided the prerequisites for this chapter: investigating how annotations can be automatically refitted when the underlying document content changes. This chapter describes an exploratory study into user expectations of annotation refitting.

This chapter covers two cycles of a study investigating user opinions of different refitting approaches. Section 5.1 describes the context and prior work for these studies. This is followed by a description of the approach used (§5.2), details on the implementation within vsInk (§5.3) and then the user study methodology (§5.4). Section 5.5 describes the main results from the study. Section 5.4 discusses the key findings and their significance for improving refitting. Finally, section 5.7 ties these results into the main theme of the thesis.
The study described in this chapter was conducted with the approval of the University of Auckland Human Participants Ethics Committee. A copy of the approval letter is in Appendix A.

Some of the results from study have been published previously in Sutherland et al. (2016b).

## 5.1 Introduction

Refitting freeform ink annotations is difficult. Prior literature only lists a limited number of refitting algorithms, with the most common approach being to do nothing (leave the annotation in their original form.) While this is simple to implement it carries the possibility that the annotations will lose their meaning. To retain their meaning, annotations need to refit in response to changes in the underlying context: changing their form to match the new content. Previous studies demonstrated that the refitting approach varies by annotation type (Golovchinsky and Denoue, 2002, Bargeron and Moscovich, 2003).

This chapter extends the work by Golovchinsky and Denoue (2002) and Bargeron and Moscovich (2003) in three ways. First, by investigating whether annotations need to be refitted. Second, by investigating alternate ways to refit annotations. And third, by adding an additional class of annotations, connectors, with its own approaches for refitting.

For this chapter the following hypotheses are proposed:

**H5**: Users would prefer to have their annotations automatically refitted when the underlying program code changes.

**H6**: There would be a preferred style of refitting for each class of annotation.

In addition, the following sub-questions were asked:

**RQ2.2.1**: What does the user think of refitting their annotations?

**RQ2.2.2**: What is their preferred approach for adapting the following classes of annotation:

(a) Horizontal lines;
(b) Vertical lines;
(c) Enclosures;
(d) Connectors?

The next section outlines the general approach used to answer these questions.
Chapter 5. User Study 3

TABLE 5.1: Categories of annotation types investigated in this chapter and examples of each category.

<table>
<thead>
<tr>
<th>Group</th>
<th>Annotation Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal Line</td>
<td>Underlines, highlights and scratch-outs</td>
</tr>
<tr>
<td>Vertical Line</td>
<td>Margin bars and braces</td>
</tr>
<tr>
<td>Enclosures</td>
<td>Circular and rectangular bounding boxes</td>
</tr>
<tr>
<td>Connectors</td>
<td>Call-outs and arrows</td>
</tr>
</tbody>
</table>

5.2 The Approach

The primary focus in this study is the users’ opinions of refitting. To investigate this, several different refitting algorithms were implemented in vsInk. These algorithms were evaluated by users over two rounds of studies. The refitting algorithms were based on previous research (Golovchinsky and Denoue, 2002, Bargeron and Moscovich, 2003), ideas from a focus group of researchers in the area of ink sketching and feedback from the participants.

In the first round of user studies, twelve different algorithms were investigated. These algorithms were suggested by previous research and the focus group. The participants were asked to read and annotate a short section of program code, as prior research indicates users react better to modifications of their own annotations (Bargeron and Moscovich, 2003). After reading, the underlying text was changed and the participants evaluated each algorithm.

In the second round of studies, fourteen different algorithms were investigated. These included preferred algorithms from the first round and new algorithms based on user feedback. This round used the same protocol as the first round.

5.3 Implementation

Following the strategy used by previous studies, annotation types were grouped into four categories. I hypothesised users would prefer the same adaptation process for each annotation type in a category. Table 5.1 shows the categories, based on prior research (Golovchinsky and Denoue, 2002, Bargeron and Moscovich, 2003).

In this study the RATA.Gesture recogniser (Blagojevic et al., 2010) was used in vsInk with a verification step as this allowed changing the annotation if it was
incorrectly classified\textsuperscript{1}. The recogniser for this study identified horizontal lines (highlights and underlines), vertical lines (braces and margin bars), enclosures (circles), connectors and text. While text annotations were anchored in the document, they were not refitted in any way.

5.3.1 Horizontal Lines

Five ways of refitting horizontal line annotations were implemented (see Figure 5.1). The first three ways are based on where the annotation is split.

**Unmodified:** the annotation is not split at all (see Figure 5.1b.). The annotation is associated with the first underlying character.

**Word Boundary:** the annotation is segmented against each word (see Figure 5.1c.).

**Character Boundary:** the annotation is segmented against each character (see Figure 5.1d.).

Each annotation is split based on where characters were added between the boundaries; with the annotation staying relative to the first associated character. If the text is split between the segments the annotation will stay with the previous anchor. Previously the only algorithm for refitting horizontal line annotations

\textsuperscript{1}See Chapter 4 for details on this approach.
was to split them using word boundaries (Golovchinsky and Denoue, 2002, Bargeron and Moscovitch, 2003).

The remaining two algorithms use character boundary splitting as the starting point.

**Split with lines:** vertical dashed lines are added at the split locations (see Figure 5.1e.) This dashed line is semi-transparent and extends above and below the annotation for half the character height.

**Split with stretch:** the annotation is stretched so the previous segment fills the gap (see Figure 5.1f.). This gives the illusion that the annotation itself has been stretched to fill the gap.

### 5.3.2 Vertical Lines and Enclosures

Eight algorithms were implemented for refitting vertical line and enclosure annotations. The first two have been implemented previously and the remaining six are new. Figure 5.2 shows these algorithms.
These algorithms only modify the annotation vertically. Another class of modifications is to add text within the annotation that would skew the annotation horizontally. To reduce the complexity of both the implementation and the evaluation, it was decided to only handle vertical changes\(^2\).

Vertical lines and enclosures were grouped together as both classes span multiple lines. The hypothesis for this decision is users would prefer similar algorithms for both categories of annotation. Figure 5.2a. shows the original unmodified annotation and the underlying context.

**Unmodified:** the annotation was left as it was originally (see Figure 5.2b.). This is the algorithm typically used in the majority of implementations (e.g. Sutherland and Plimmer, 2013) but no one has reported on the desirability of this algorithm.

**Whole stretch:** the entire annotation is scaled (see Figure 5.2c.). This is the algorithm used in XLibris (Golovchinsky and Denoue, 2002) and Callisto (Bargeron and Moscovich, 2003). In this algorithm, the annotation is associated with the lines at the top and bottom of the annotation. The annotation is scaled vertically using the distance between the top and bottom of the annotation.

**Simple split:** the annotation is split into segments without any indicators (see Figure 5.2d.). The annotation is segmented so each segment is associated with a single line of text. When a new line is added, the annotation is split and the gap is left unfilled.

**Split with vertical line:** the annotation is split into segments with a vertical line between segments (see Figure 5.2e.). This line is dashed and semi-transparent so it is different from the rest of the annotation. The line is always vertical: it directly joins the last point of the previous stroke with the first point of the next stroke.

**Split with horizontal line:** the annotation is split with a horizontal line added (see Figure 5.2f.). For enclosures, the line goes from the outer-most point on the left to the outer-most point on the right. For vertical lines, the line extends to both sides of the line by the equivalent of two character spaces. For both variations the line is dashed and semi-transparent.

**Split with horizontal dashes:** similar to the previous algorithm with both enclosures and vertical lines having a short line extending to the sides of the

---

\(^2\) Including horizontal changes would mean needing to evaluate three conditions: vertical-only, horizontal-only and horizontal-and-vertical. Each condition would need to be evaluated independently, otherwise there may be confounding influences between these changes.
stroke (see Figure 5.2g.).

**Split with adaptative line**: combines ‘Split with vertical line’ and ‘Split with horizontal line’ (see Figure 5.2h.). If the segments surrounding the split are nearly vertical, a vertical joining line is added; otherwise horizontal lines are added.

**Stretch segment only**: the annotation is split into two segments and the top segment is scaled to fill the gap (see Figure 5.2i.). Unlike the whole annotation stretch, only the segment above the gap is scaled.

For all split algorithms, the annotation was segmented based on lines. When the stroke crossed the line, it was segmented into two. To ensure the annotation looked smooth, an extra point is extrapolated at the line boundary. This point becomes the final point of the old segment and the first point of the new segment.
5.3.3 Connectors

Four algorithms for refitting connectors were implemented: none have been investigated previously (see Figure 5.3). In vsInk, adding a connector annotation automatically adds an associated note. This note is styled as a yellow post-it note. Any strokes added within this note are automatically grouped together as a complex annotation (e.g. text or drawing).

Annotations have two anchor points: one at the start and one at end of the stroke. The primary anchor point is the start point of the stroke. This allows for both vertical and horizontal changes in the position of the anchor.

**Unmodified**: the annotation is unchanged by any context changes (see Figure 5.3b.). This causes the annotation to move together with the primary anchor point. When the associated note is on a different line the two annotations become separated as space is added between their anchor points.

**Whole stretch**: the annotation is stretched vertically between the start and end points (see Figure 5.3c.). The annotation is only stretched vertically which can result in some distortion.

**Points stretch**: the annotation is stretched on an axis between the start and end points (see Figure 5.3d.). This tends to result in a smoother scaling of the line.

**Dynamic stretch**: switches between ‘Unmodified’ and ‘Points stretch’ depending on where the end point is relative to the associated note. When the end point is within the note boundary the annotation is unmodified. When the end point moves beyond the boundary of the note the end point ‘sticks’ to the top, or bottom, of the note and ‘Points stretch’ is used.

5.4 User Study

To evaluate people’s reactions, two cycles of the study were performed. The primary focus was what people would think of the different adaptations and which they would prefer for each annotation type. The two studies followed the same protocol, based on the study by Bargeron and Moscovich (2003). The only difference between the two studies was the algorithms used (see Table 5.2).
Table 5.2: Refitting algorithms used in each study.

<table>
<thead>
<tr>
<th>Annotation Class</th>
<th>Algorithm</th>
<th>Study One</th>
<th>Study Two</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal Lines</td>
<td>Unmodified</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Split by word</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Split by character</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Split with lines</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Split with stretch</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Vertical Lines</td>
<td>Unmodified</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Whole stretch</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Simple split</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Split with vertical line</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Split with horizontal line</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Split with horizontal dashes</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stretch segment only</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Enclosures</td>
<td>Unmodified</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Whole stretch</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Simple split</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Split with vertical line</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Split with horizontal line</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Split with horizontal dashes</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stretch with adaptive line</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Stretch segment only</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Connectors</td>
<td>None</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Vertical Stretch</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Points Stretch</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Dynamic Stretch</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

A Microsoft Surface tablet was used for both rounds of testing. The reading was all done in vslnk (Sutherland, 2012). Morae (TechSmith, 2015) was used to record the screen to allow for analysis after the testing.

5.4.1 Procedure

Each participant performed the task in a controlled environment. Prior to the task each participant was welcomed, had the process explained to them (including gaining consent) and filled in a short questionnaire on their reading and annotating background.

After the questionnaire was completed, vslnk was demonstrated. The participants were then given 5 minutes to familiarize themselves with the software
and ask any questions. If desired, the participant could finish this part early. The participants were then given 15 minutes to read a small program. The instructions were to read and annotate the code so they could explain it to another programmer. They were specifically instructed not to change the code. During the reading time, the participant was observed.

After the reading task, the researcher selected several annotations of each category. If there were insufficient annotations in a category the participant was asked to add some more. The researcher modified the text underneath each annotation and showed the participant each way the annotation could be refitted. The participant then rated each adaptation using a five point Likert scale. The statement was “I like how the annotation appears after it has adapted to the change in the code”: one meaning strongly agree through to five meaning strongly disagree. The participant was then asked to choose their most and least preferred algorithms for the adaptation.

The underlying text was modified in specific ways for each class of annotations.

**Horizontal lines:** Additional text was added in the middle of the annotation.

**Vertical lines and enclosures:** New lines were added in the middle of the annotation.

**Connectors:** New lines were added in between the start and end points of the connector. Additional lines were added until the connector detached from the associated note.

After reviewing the annotations, there was a short interview with the participant. Each participant was asked to share their views on refitting and how annotations might be useful during programming. Finally, the participant was thanked for their time, which concluded the trial.

### 5.4.2 Analysis

For each participant, the questionnaire, ratings and ranking data were added into a database. The data were entered in an anonymous format so the participants could not be identified. The data were analysed using R 3.2.2 (R Core Team, 2015, Bryer and Speerschneider, 2015).

The discussions during the review and interview stages were transcribed. The scripts for each annotation class were then collated and analysed as a group. The analysis consisted of reading each script and finding common comments.
To compare results from the Likert scales, a “liking” score was calculated for each adaptation algorithm using the following formula:

$$\text{score}_{\text{algorithm}} = \frac{\sum_{i=1}^{n} (5 - \text{score}_i)}{n}$$

where \(\text{score}_{\text{algorithm}}\) is the score for the algorithm, \(n\) is the number of annotations adapted and \(\text{score}_i\) is an individual liking score.

### 5.5 Results

The adaptations were evaluated over two rounds of the study with different participants in each round. The protocol and task was almost the same for both rounds: the only difference was the algorithms evaluated. Table 5.2 lists the algorithms used in each round.

#### 5.5.1 Round One

In the first round, there were seven participants. All participants were familiar with reading program code and could understand the code provided. All participants reported reading code on a regular basis (at least daily). All participants were used to annotating while reading; although only four reported annotating program code. During the interview, this was discussed. One participant would occasionally print out code on paper and add freeform annotations. The other three added comments as ‘annotations’ to the code\(^3\). The remaining three programmers did not make any form for annotation for understanding the code\(^4\).

There were 102 annotations collected in round one (34 single line, 19 vertical line, 19 enclosure and 30 connector). Figure 5.4 shows the results from the first round. There was no evidence of any difference in the number of annotations collected between the participants \((F(6,21) = 0.74, p = 0.62)\). Table 5.3 shows the order of algorithms by participant preference.

---

\(^3\) One participant specifically mentioned he deleted these comments prior to committing the code to source control.

\(^4\) One participant specifically mentioned he never adds any non-code text to code files as they “make the code messy.”
Figure 5.4: Participants’ agreement with the statement “I like how the annotation appears after it has adapted to the change in the code” for each adaptation algorithm in round one. 

- **a.** horizontal lines; 
- **b.** vertical lines; 
- **c.** enclosures; 
- **d.** connectors. 

Numbers on the left are the percentage of responses coloured red, numbers in the middle are the percentage of responses coloured yellow, numbers on the right are the percentage of responses coloured green.
TABLE 5.3: Algorithms ordered by preference in Study One. Count is the number of algorithms rated as the preferred algorithm for the adaptation.

<table>
<thead>
<tr>
<th>Annotation Class</th>
<th>Algorithm</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal Lines</td>
<td>Split by word</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>Split by character</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Unmodified</td>
<td>7</td>
</tr>
<tr>
<td>Vertical Lines</td>
<td>Split with vertical line</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Whole stretch</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Split with horizontal line</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Stretch segment only</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Simple split</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Unmodified</td>
<td>0</td>
</tr>
<tr>
<td>Enclosures</td>
<td>Whole stretch</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Unmodified</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Split with horizontal line</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Split with vertical line</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Stretch segment only</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Simple split</td>
<td>0</td>
</tr>
<tr>
<td>Connectors</td>
<td>Points Stretch</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Vertical Stretch</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>1</td>
</tr>
</tbody>
</table>

For horizontal lines, the algorithm with the highest “liking” score was segmenting by word (3.09) and then segmenting by character (2.26): doing nothing had the lowest score (1.38). This matches the preferences: segmenting by word was chosen as the most preferred approach ($n = 17$) and doing nothing the least ($n = 7$). However there were some circumstances where these were reversed. Users preferred the line to stay with the associated syntactical unit. However, if it did not, then the users preferred that the annotation was not modified.

For vertical lines, doing nothing had the lowest score (0.58) and was never chosen as the preferred approach. This may be because vertical lines are context specific, with both the top and bottom position of the line being significant. Splitting the annotation without any visualisation also had a low score (1.42) and was never chosen as the preferred approach. Without the visualisation several participants mentioned this algorithm appeared to make two separate annotations instead of one split into two.

Split with vertical line had the highest score (3.26), followed by stretching the whole annotation (2.84) and then by stretching the segment (2.53). Split with
vertical line was also chosen as the most preferred approach (n = 7).

While enclosure annotations use the same adaptation routines as vertical line, the results are less consistent. One difference is the ‘unmodified’ algorithm had a higher score than vertical lines (1.21) and was not the lowest score (simple split with 1.16). The reason for this is in some situations the enclosure is a form of single line annotation. When asked, the participants stated they expected these enclosures to be refitted in a similar way to underlines (i.e. single line annotations).

Another difference is the approach with the highest score was the whole stretch algorithm (2.47) with split with vertical line being next (2.37). The reason for this difference appears to be that location of the code change relative to the annotation makes a difference. If the segments of the annotation near the change were near vertical, then a vertical line was preferred; otherwise stretching the whole annotation was preferred.

For connector annotations, all participants preferred the annotation should stay with the associated note. The approach with the highest score was stretching between the points (3.50) followed by stretching vertically (2.7). The preferred algorithm counts also match these results: points stretch was the preferred algorithm (22 times) and vertical stretch (7 times). Many participants mentioned they could not tell the difference between the two approaches most of the time. Based on observations, the main time the stretching was different was when the annotation was not a straight line (for example when an arrowhead was included on the connector). Another point raised in the interviews, is most participants liked the unmodified algorithm until the note became detached from the annotation. At this point they mentioned the meaning of the connector was lost.

5.5.2 Round Two

In the second round, there were ten participants. All were professional developers. All participants reported reading code on a regular basis. Only three of the participants reported annotating program code.

There were 165 annotations collected in this round (59 horizontal line, 31 vertical line, 30 enclosure and 45 connector annotations). Figure 5.5 shows the results from this study. There was no evidence of any difference in the number of
FIGURE 5.5: Participants’ agreement with the statement “I like how the annotation appears after it has adapted to the change in the code” for each adaptation algorithm in round two. a. horizontal lines; b. vertical lines; c. enclosures; d. connectors. Numbers on the left are the percentage of responses coloured red, numbers in the middle are the percentage of responses coloured yellow, numbers on the right are the percentage of responses coloured green.
annotations collected for each participant ($F(9, 30) = 0.14, p = 0.99$). Table 5.4 shows the order of algorithms by participant preference.

For horizontal line annotations, doing nothing again had the lowest “liking” score (1.46) and none of the participants chose it as their preferred algorithm. The times participants did rate it below three on the likert scale, the annotation was either long enough or the change small enough that the annotation remained underneath the relevant words. During the interview, participants revealed the only time doing nothing was acceptable was when the relevant words were still underneath.

Of the remaining three approaches, stretching the annotation had the highest score (3.02) followed by splitting and adding a horizontal line (2.61). Stretching was chosen as the preferred approach most often ($n = 32$), however, these numbers do not show a consistent trend observed: the participants preferred one or the other of the algorithms. Thus, the participants can be divided into two groups: those who liked to see the annotations had been adapted and those who preferred that the annotations stayed natural looking.
The main reason mentioned for preferring the stretch algorithm is the annotations look more natural and match the original annotation (six participants). Comments from these participants included “this looks like what I expect”, “I like how it maintains the look of the line” and “it still feels like it is one annotation”. In contrast, the main reason for liking the visualisation is it shows that the annotation has been split (four participants). One participant mentioned “without the dashed lines I would have had no idea the annotation has been split”. Another participant stated “it shows clearly where the change is”.

For vertical lines, there is a similar split between stretching and adding dashed lines. For the dashed lines, the algorithm with the highest score was to split the annotation and add a vertical dashed line (3.28), with stretching having a lower score (2.72). From the interview, it emerged that the participants preferred vertical lines over horizontal lines as they showed the segments of the annotation belonged together.

Finally, all the participants stated the two stretch algorithms were very similar. During the review stage they often found it hard to tell them apart. However, for some types of vertical lines it was more obvious that only part of the annotation had been stretched. In this case most of the participants (eight) preferred the whole stretch as it maintained the natural look of the annotation.

Again, the preferences for enclosure adaptations are less obvious, for similar reasons to round one. In this study, some participants still preferred no adaptation in certain scenarios (when the enclosure is associated with a line), and this was the preferred algorithm for two cases; otherwise this algorithm still had the lowest score (0.97). However if the enclosure is a multiple line annotation then it should be modified as the top and bottom positions have significance.

A new algorithm in this round was the adaptive algorithm. In this approach the annotation refitted based on the surrounding segments (see Figure 5.2h.). This algorithm had the second lowest score (2.00) and was never chosen as the preferred algorithm. According to the participants the main problem is the results were unpredictable. One participant stated this was “confusing with a vertical line on one side and horizontal on the other.” Likewise, stretch segment was also less liked, although less so than the dynamic algorithm. The problem is it would sometimes distort the annotation, losing the natural shape of the annotation.

Of the remaining algorithms, split with vertical lines had the highest score (3.03),
The Curious Case of Annotation Adaptation

Table 5.5: Calculated “liking” scores for each algorithm. Algorithms not included in a study are marked as n/a. Algorithms are ordered by their score in study two, if included, then by their score in study one.

<table>
<thead>
<tr>
<th>Annotation Class</th>
<th>Algorithm</th>
<th>Study One</th>
<th>Study Two</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal Lines</td>
<td>Split with stretch</td>
<td>n/a</td>
<td>3.02</td>
</tr>
<tr>
<td></td>
<td>Split with lines</td>
<td>n/a</td>
<td>2.61</td>
</tr>
<tr>
<td></td>
<td>Split by word</td>
<td>3.09</td>
<td>2.03</td>
</tr>
<tr>
<td></td>
<td>Unmodified</td>
<td>1.38</td>
<td>1.46</td>
</tr>
<tr>
<td></td>
<td>Split by character</td>
<td>2.26</td>
<td>n/a</td>
</tr>
<tr>
<td>Vertical Lines</td>
<td>Split with vertical line</td>
<td>3.26</td>
<td>3.28</td>
</tr>
<tr>
<td></td>
<td>Whole stretch</td>
<td>2.84</td>
<td>2.72</td>
</tr>
<tr>
<td></td>
<td>Split with horizontal dashes</td>
<td>n/a</td>
<td>2.45</td>
</tr>
<tr>
<td></td>
<td>Stretch segment only</td>
<td>2.53</td>
<td>2.34</td>
</tr>
<tr>
<td></td>
<td>Split with horizontal line</td>
<td>1.63</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Simple split</td>
<td>1.42</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Unmodified</td>
<td>0.58</td>
<td>n/a</td>
</tr>
<tr>
<td>Enclosures</td>
<td>Split with vertical line</td>
<td>2.16</td>
<td>3.03</td>
</tr>
<tr>
<td></td>
<td>Whole stretch</td>
<td>2.47</td>
<td>2.88</td>
</tr>
<tr>
<td></td>
<td>Stretch segment only</td>
<td>2.05</td>
<td>2.59</td>
</tr>
<tr>
<td></td>
<td>Split with horizontal dashes</td>
<td>n/a</td>
<td>2.22</td>
</tr>
<tr>
<td></td>
<td>Split with adaptive line</td>
<td>n/a</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>Unmodified</td>
<td>1.21</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>Split with horizontal line</td>
<td>1.84</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Simple split</td>
<td>1.16</td>
<td>n/a</td>
</tr>
<tr>
<td>Connectors</td>
<td>Points Stretch</td>
<td>3.50</td>
<td>3.33</td>
</tr>
<tr>
<td></td>
<td>Dynamic Stretch</td>
<td>n/a</td>
<td>3.20</td>
</tr>
<tr>
<td></td>
<td>Vertical Stretch</td>
<td>2.70</td>
<td>2.51</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>0.60</td>
<td>0.38</td>
</tr>
</tbody>
</table>

followed by whole stretch (2.88). These two algorithms were chosen as the preferred algorithm the most (vertical lines: $n = 13$, stretch: $n = 10$). Again, there were some participants who preferred the natural appearance and others who preferred to see that the annotation had split.

One trend was observed across the three categories: the participants preferred the same ‘style’ of refitting. If the participant preferred horizontal lines stretched, then they would also prefer vertical lines and enclosures stretched. The same pattern also applies for adding lines. The participants can be categorised as belonging to one of two groups: those who preferred stretching (natural appearance) and those who preferred an indicator the annotation had refitted.
Again, all participants agreed that connector annotations should be refitted. The participants mentioned they liked the new dynamic algorithm for two reasons. First, it stayed in the original form when relevant and only refitted as necessary. Second, it did adapt when necessary in a way that both preserved the meaning and looked natural. This was the preferred approach for refitting connectors \((n = 24)\). In this study, the points algorithm had the highest score (3.33) and was the second most preferred algorithm \((n = 15)\).

There were also some suggested enhancements for connector refitting that came from the interviews. One participant suggested that “the note annotation could be considered part of the connector annotation.” That is, there should only be one anchor for the combined connector and note, with the note staying fixed to this anchor. Another common request for refitting connectors, is the new connector should avoid the underlying text. One participant stated it as “I would normally add a line in the gap between the lines, having the line go over the text is not how I would do it.” Three other participants made similar comments. Also, four participants mentioned they would like the connector to stay in the white space rather than just stretch.

Table 5.5 shows the calculated “liking” scores for all algorithms in both studies.

### 5.6 Discussion

This study set out to determine if and how freeform annotations should be refitted. The first hypothesis is participants would prefer their annotations refitted. For most annotations this was true but there was a consistent set of exceptions. First, the underlying context needed to change enough for the refitting to be meaningful, otherwise the participants saw no value in refitting. A common example, was underlines where the underline extended beyond the end of the line. Second, some enclosure annotations should be treated similar to horizontal line annotations. While it is posited these enclosures should adapt in a similar way to horizontal lines this adaptation was not investigated.

The meaning of annotations has to be preserved, and as long as the change to the text preserves the meaning, then the users saw no reason to refit. For example, underlines only have meaning when associated with the underlined words. If the annotation moved away from those words, then it would become either
meaningless or confusing (as one participant stated ‘I would wonder why I had only underlined that part, not the whole line’).

The second hypothesis is there would be a preferred approach for each annotation category. The preferred approach would preserve the meaning of the annotations. To evaluate this hypothesis the annotation types were initially grouped into four categories. Based on previous studies (Golovchinsky and Denoue, 2002, Bargeron and Moscovich, 2003), the lines were split into vertical or horizontal. The results indicate the participants preferred similar approaches for both categories, suggesting these could be combined into a single category. Further work needs to be done to establish whether both types of lines should adapt in the same way. A future area of investigation is on what should happen when horizontal line annotations wrap to a different text line.

One disagreement in previous studies, was how to categorise enclosures. Callisto treated enclosures the same as underlines (Bargeron and Moscovich, 2003) while XLibris treated them as vertical lines (Golovchinsky and Denoue, 2002). The results show there are two categories of enclosures. Short enclosures around one or more words on the same line are similar to horizontal lines. Taller enclosures that span multiple lines are in their own category with different algorithm preferences from either type of line.

The results of this study suggest some general recommendations for each category. First, an annotation, regardless of class, should be recognisable as a single annotation. This means when splitting an annotation some visualisation must be added to show the annotation has split. Second, stretching an annotation is acceptable as long as the annotation is not distorted. This means for lines it is acceptable to stretch a single segment but for enclosures it is better to stretch the entire annotation.

In addition, for connectors it is important any associated annotation remains linked to the connector. In the study connectors were the only annotations investigated with an associated annotation, but there were other examples of where annotations should be linked. One common example is a vertical line (emphasizing a range of lines) followed by a connector. These should also be linked and the association maintained. However, these were not included in the investigation as this involves challenges of how to group together annotations (see Chapter 2).

Some participants also raised the issue of text avoidance for connectors. In this
implementation connectors do not avoid covering text. One participant pointed out they add connectors in the whitespace and they would expect any refitting to respect this. While text avoidance has not been investigated for annotations, it has been investigated when manipulating connectors on graph diagrams (Plimmer et al., 2009). The same concept should work for annotations with using text as nodes. This would require any implementation to be more aware of the underlying text and how to re-route connectors in relation to it.

In addition, the results of this study indicate there is not one but two preferred sets of refitting. One group of participants preferred algorithms that kept the original look and feel: this group is labelled ‘natural’. The second group prefers to see their annotations were refitted. This group is labelled ‘indicator’ as the preferred algorithms indicate not only the underlying code has changed but where the change was.

In natural mode annotations are refitted by stretching, while in indicator mode the annotation is split and a visualisation added. Any implementations should allow the user the choice of mode. Potentially, users may also want to swap between modes but this was not investigated in this study.

While program code was used as the context of the investigation, these findings are likely to be similar for other forms of document. The main rationale for this is the annotations added to program code are very similar to annotations on other forms of text (see Chapter 3). This generalisation has some limits. First, program code tends to have more whitespace than other forms of prose (although there is evidence showing the benefits of adding additional whitespace for annotations - Pearson et al., 2009, Yoon et al., 2013). Second, lines in program code do not always wrap. Instead readers scroll horizontally for long lines. Therefore, this study does not extend to the effects of line wrapping on annotations.

5.7 Summary

This chapter investigated different approaches for automatically refitting annotations when the underlying context changed. Several refitting algorithms were implemented in vslnk. A user study investigated people’s reactions to these approaches. There were two rounds of the study with the implementations in vslnk being refined after each round. Users expect their annotations to refit
and the preferred refitting algorithm depends on two factors: the category of annotation and the user’s general preference.

As suggested previously (Golovchinsky and Denoue, 2002, Bargeron and Moscovich, 2003), the category of annotation is important. Four categories were explored: horizontal lines, vertical lines, enclosures and connectors. There are different preferences for each category. In addition, enclosures should be two separate categories, depending on whether they are associated with one line or multiple.

One unanticipated result was there are two different groups of users. One group prefers natural looking annotations with stretching as the preferred approach. The other group prefer some form of indication that the context underneath has changed. These users preferred splitting annotations and adding a visualisation.

The results from this study suggest the following preferences:

1. Line annotations (horizontal and vertical) should be either segment stretched or split with a visualisation;
2. Multiple line enclosure annotations should be either whole stretched or split with short perpendicular lines;
3. Connectors should be left as is when the associated note is connected; otherwise they should be stretched.

In addition, there appear to be similar characteristics between single line enclosures and horizontal lines. However, the study did not investigate whether the adaptations for horizontal lines have similar results when used on a single line enclosure.

This chapter completes the investigations of freeform digital ink using vsInk. The next chapter changes the focus onto how this work fits in with the bigger picture of freeform digital ink annotations.
Discussion
Back to the Annotation

“The most exciting phrase to hear in science, the one that heralds new discoveries, is not ‘Eureka!’ but ‘That’s funny...’”

Isaac Asimov

In the previous three chapters, I described three studies concerning freeform digital ink annotations. This chapter now discusses the results from these studies and shows how they address the main research questions for this thesis. This discussion follows the two main research questions.

Section 6.1 focuses on **RQ1.** and discusses how and why programmers annotate, and how this knowledge can be used for implementing freeform annotations within an Integrated Development Environment. Section 6.1.1 contrasts Marshall’s taxonomy of annotations (1997) with the taxonomy developed in this thesis and how the new taxonomy can guide the development of freeform annotation functionality. Sections 6.1.2 and 6.1.3 review the two main reasons why programmers added annotations. Section 6.1.2 discusses how freeform annotations may be beneficial for navigation and some possible limitations. Finally, Section 6.1.3 explores how freeform annotations could be a simple, low overhead approach for offloading information.

Section 6.2 focuses on **RQ2.** and how to extend Integrated Development Environments to support freeform digital ink annotations. Section 6.2.1 describes the two overall sequences of operations for adding annotations and the implications
of each sequence. Section 6.2.2 discusses when recognition may be needed and how this influences the design. Section 6.2.3 considers annotation refitting, especially preserving the meaning of annotations. Finally, Section 6.2.4 describes a possible design of how freeform digital ink annotations can be implemented in a natural way within an Integrated Development Environment.

Section 6.3 summarizes the key points from this chapter.

6.1 The How and Why of Programmer Annotation

Given that programmers very rarely print code on paper, the first research question investigated whether they would annotate code if it were possible. The results of the user study in Chapter 3 show that they annotate on paper when presented with printed code and a code comprehension task. The fact that programmers chose to annotate code on paper suggests that they may also choose to do so in a programming environment if the appropriate annotation tools are available. This is confirmed by comments during the second and third user studies (Chapters 4 and 5) where some participants stated they do annotate program code.

The user study in Chapter 3 also identified that annotations on program code are similar to annotations on other forms of prose. This is important as there are few studies investigating annotations on program code but there are more studies investigating annotations in other contexts.

6.1.1 The Two Taxonomies of Annotations

As Chapter 5 demonstrated, the category of annotation is important for implementing functionality. Two ways of categorising annotations are available: Marshall’s taxonomy (1997) and the taxonomy described in this thesis (see Tables 2.3 and 2.6 in Chapter 2). What is less obvious is what the differences are between these two taxonomies and the implications of these differences.

The seminal work on classifying freeform annotations is Marshall’s study of textbook annotations (1997, 1998). She identified several different archetypes of annotations. These archetypes, or classes of annotation, formed the foundation of a simple annotation classification taxonomy that can classify most types of annotation. Chapter 1 summarises her classification taxonomy (see Table 1.1).
Chapter 6. Discussion

Table 6.1: Revised Taxonomy of Annotation Types

<table>
<thead>
<tr>
<th>Type Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single line</td>
<td>Underlines</td>
</tr>
<tr>
<td></td>
<td>Highlighting</td>
</tr>
<tr>
<td></td>
<td>Enclosures around words</td>
</tr>
<tr>
<td>Multiple line</td>
<td>Enclosures around paragraphs</td>
</tr>
<tr>
<td></td>
<td>Margin bars</td>
</tr>
<tr>
<td></td>
<td>Braces</td>
</tr>
<tr>
<td>Connectors</td>
<td>Callouts/arrows</td>
</tr>
<tr>
<td>Complex</td>
<td>Text/symbols within text</td>
</tr>
<tr>
<td></td>
<td>Drawings</td>
</tr>
<tr>
<td></td>
<td>Marginalia</td>
</tr>
<tr>
<td>Commands</td>
<td>Commands</td>
</tr>
</tbody>
</table>

These classes are based on where the annotation is and how obvious its meaning is (its explicitness).

While the location is easy to determine computationally, the explicitness is not. Therefore, in the literature review an alternate classification system is proposed (see Table 2.3). This classification system was evaluated during the user studies described in Chapters 3 to 5. Table 6.1 presents a revised taxonomy based on the results from these studies.

One important reason for the difference in taxonomies is the rationale behind them. In Marshall’s study (1997), she was interested in seeing how people annotated on paper. In contrast, in the studies in this thesis, the focus was investigating how annotations have been implemented in systems. While there is an overlap between the two taxonomies, each has a different focus. For example, Marshall’s taxonomy breaks down different types of text annotations, while in the revised taxonomy, they are a single classification.

Each taxonomy uses different details to classify the annotation. Marshall’s taxonomy (1997) uses the location of the annotations and their meaning. The revised taxonomy uses the geometric characteristics of the annotations and the relationship of the annotations to the underlying text. Thus, the revised taxonomy is easier to implement in an automated system as it does not need to infer the implicit meaning behind the annotations.

One important implication of the revised taxonomy is annotations in the same category can be adapted in the same way. For example, single line annotations
can all be refitted by stretching the annotation horizontally\(^1\). While this research only explored refitting (see Chapter 5), I hypothesise the other adaptation operations would also use these categories.

Another implication is how these categories inform the adding operations. vs\(\text{Ink}\) implemented a different stroke processor for each category of annotation (see Appendix B). In contrast, in an earlier work, there was a processor for each type of annotation (Sutherland, 2012), resulting in more duplication. In theory, it may be possible to combine some of these categories (i.e. horizontal and vertical lines) based on the type of functionality supported.

In terms of implementing annotation functionality, there is another implication of the revised taxonomy: the system does not need to infer the meaning of the annotation. This is important as it is currently very difficult to automatically infer the meaning of an annotation.

### 6.1.2 Annotations for Navigation

Navigation was the main reason the participants added freeform annotations in the first user study (see Table 3.3). However, an Integrated Development Environment already provides a richer set of navigation aids than paper. Reviewing Table 3.3, there are four sub-reasons for adding navigation annotations. The most common sub-reason is to *emphasize a feature of the code for future reference*, followed by *adding a reference*, *emphasizing the code structure* and *emphasizing a significant feature*.

Of these four sub-reasons, *adding a reference* and *emphasizing the code structure* are provided by existing functionality in Integrated Development Environments. For example, most Integrated Development Environments provide some form of syntax highlighting which allows the reader to see the code structure, and tools like *Solution Explorer*, *Goto Definition* and *Peek Definition* in Visual Studio provide easy referencing to other locations of the code.

The other two sub-reasons are user-specific: these are places that the reader has emphasised as being important during the task. Therefore, being able to find these annotations is useful. This is the sort of annotation navigation that *CodeAnnotator* (Chen and Plimmer, 2007) and the original version of *vs\(\text{Ink}\)* (Sutherland and Plimmer, 2013) allows. While not formally recorded, this was

\(^1\)This is a single example. Chapter 5 describes refitting in more detail.
the main reason for navigation annotations observed during the second user study.

In addition, freeform annotations may provide value when retracing another programmer’s path through the code. There are a variety of reasons why a programmer may want to retrace: for learning, debugging, documenting, etc. (DeLine et al., 2005) As the programmer is following the original person’s path they do not need to find new areas nor is spatial memory as important. As freeform annotations are a low-overhead means of marking a document while reading (O’Hara and Sellen, 1997), the original reader can subconsciously add annotations for another person to follow. While these annotations might be added deliberately, it is possible another person would benefit from personal annotations. Marshall (1997) documents instances where students valued other annotations as a way of indicating what is important.

The interface for navigation would need to provide for these different forms of navigation. There are two basic forms of spatial navigation: within a document and across multiple documents. Within a document would allow a reader to quickly scan through the document and see where it has been annotated. This would be useful for when the reader knows which file to view (Sutherland and Plimmer, 2013). In contrast, the second form, across documents, would allow a reader to see which documents have been annotated (Chen and Plimmer, 2007). As well as the spatial displays, the annotations could be displayed on a timeline to allow a second reader to see when and where the original person added the annotations.

### 6.1.3 Annotations for Offloading Information

The previous section discussed using annotations for navigation. The other two reasons why the participants added annotations are related to Information Offloading (Working Information and Information Sharing, see §3.3.3). Readers only have a limited capacity for storing items in memory: once these “memory slots” are full new items can only be added by “forgetting” something or combining items together (Rogers, 2004). This is illustrated by a comment from one of the participants: by the end of the reading task he felt like he was “juggling several plates on sticks”. Using annotations offloads information from a person’s working memory onto the page. Offloading information from working memory
Information offloading relates to external cognition in two different ways (Scaife and Rogers, 1996b, Rogers, 2004). First, it is a way of Re-representing Information. Re-representation is changing the information so it contains the same details but in a different format. This then allows the person to see the information in a different, potentially easier, way. Second, it is a way of spatially constraining cognition. In this form, the constrained area helps to trigger cognitive processes. Both ways are useful for reducing how much information a person needs to hold in their working memory.

Different types of annotation potentially help information offloading in different ways. For example, notes are useful for re-representing the information but are less useful for spatial constraining. In contrast, highlights and underlines are better for spatial constraining. The following are some examples of how Information Offloading assists with comprehension, constrained to what is currently available in an Integrated Development Environment.

Notes allow the reader to jot down key points they are reading. Writing a summary of the key points can help the reader discern what is important in the section they are reading and how it related to other parts. It can also act as a commentary for future readers (Agosti et al., 2007). Comments are an example of inline notes that are implemented in most programming languages. In addition, there are a variety of tools that allow adding notes to alternate representations of the code (e.g. Sillito et al., 2005, Bragdon et al., 2010, DeLine et al., 2012). One limitation with these tools is they force the user to use a keyboard for entering the text. As mentioned earlier, using a keyboard for annotations is more cognitively demanding than using a pen (O’Hara and Sellen, 1997). This, in turn, can result in a reduction of comprehension (Jackel, 2014). In contrast, writing an annotation with a pen is not as demanding (O’Hara and Sellen, 1997). The reader can remain focused on the text they are reading. In addition, placing the notes in close proximity to the content allows the note and content to complement each other with minimal work (Jackel, 2014).

Another form of annotation is drawings and diagrams. These are a form of re-representation that can expose important information in a condensed form (Rogers, 2004). Current Integrated Development Environments only provide a limited ability for drawing. A typical code editor only allows the user to add text (although the editor itself may display graphical elements.) If the user wants to
add a drawing they need to change to a separate diagramming tool. As this is both a mode change (from reading to drawing) and an interruption (Parnin and Rugaber, 2012), it increases the person’s mental workload. In contrast, the ability to draw on the code with freeform ink is much easier. There is no need to switch between different modes and there is no interruption to their reading flow.

An example of an annotation that assists with spatial constraining is connectors. A connector is a link between two or more items. They allow a reader to quickly move between different but linked locations. Again, current Integrated Development Environments have limited implementations of connectors. Most connectors are implicit (where is the note in relation to the code?) Explicit connectors need to be added in a separate diagramming tool (e.g. Bragdon et al., 2010, DeLine et al., 2012). Again, this may increase the mental load and interrupt the flow of reading. Freeform annotations overcome both of these limitations. In addition, because freeform annotations are less constrained, it is possible to add alternate forms of connector, such as joining a single note to multiple locations (Tashman and Edwards, 2011a,b).

Thus far, we have seen how annotations could be useful for code comprehension and some of the limitations of current Integrated Development Environments. The next paragraphs explore some benefits of using freeform digital ink annotations for program code.

The ability to offload information is important to programmers (this is supported by Lichtschlag et al., 2014). But it needs to be implemented in a way that does not increase the load on either mental processing or working memory. One challenge with many development tools is they increase both, and thus are abandoned over time (Maalej et al., 2014). I posit that freeform annotations are a tool that could fit into this space for three reasons.

First, using a pen to annotate is a much lower mental overhead than using a keyboard (O’Hara and Sellen, 1997, Morris et al., 2007).

Second, it allows the annotator to annotate in their own idiosyncratic way without adding extra constraints (Marshall, 1997).

Third, keeping the annotation and the code close together is important (Jackel, 2014). If the annotation is close to the code, it provides spatial constraints for understanding both the annotation and the code. If the annotation becomes separated from the code, then it potentially loses value or increases the mental effort to understand it.
By understanding the different aspects of external cognition as they relate to code comprehension, we can target specific functionality. For example, if we focus on re-representation then we need to ensure our tools allow unconstrained drawing.

This section focused on two forms of information offloading (re-representation and spatial constraining) and discussed them in light of how they can help code comprehension. The next section builds on what has been alluded to earlier: how to implement freeform ink annotations in an Integrated Development Environment.

6.2 Annotations in a Digital Environment

The prior section discussed how and why programmers annotate. This included a comparison of a new taxonomy with a different focus to Marshall’s (1997) taxonomy, and how annotations help with navigation and information offloading. This section focuses on some technical issues of extending Integrated Development Environments to support freeform digital ink annotations.

6.2.1 Stringing Operations Together

One important finding of this research is the operations for adding annotations and their sequence. While previous studies show a wide variety of sequences, the two main sequences are diagrammed in Figures 6.1 and 6.2. These show a dichotomy in the sequence of steps in adding annotations. One approach is to process each stroke only once, with the processing happening when the stroke is added. The other approach is to re-process every stroke whenever a new stroke is added. Each approach has its benefits and limitations.

In the recurrent sequence (see Figure 6.1) each operation is performed every time a new stroke is added. This is because the classification information from the first operation is used in the following operations. This potentially improves the accuracy of these operations. In addition, because each annotation is classified as each new stroke is added, it can improve the accuracy of classification itself. As an example, we often write strokes out of sequence (e.g. the dot in an ‘i’ or the horizontal line in a ‘t’). As the strokes are drawn out of order it can be difficult to use a one-off classifier to identify the type of stroke. But, using the
recurrent classifier we have all the past strokes available and can use these to make a better classification.

This improved accuracy comes at a cost in processing time. As every stroke
is reclassified and they can use all the previous strokes, the computation cost is \(O(m^2)\). Thus, as the number of annotations increases, the cost can increase exponentially. This, in turn, means a recurrent approach may become infeasible above a certain number of strokes. There are some approaches that can be used to reduce the computational cost.

First, rather than including all strokes, the processing may be limited to a smaller set of strokes. Including just the strokes on a single page/screen is one example of this optimisation. Another example might be just comparing each stroke to its closest neighbour (although this would still require calculating a distance between every pair of strokes). This could be combined with storing the calculated distances between each pair and only calculating the distances between the new stroke and existing strokes.

Another optimisation approach would be to use a reduced set of features when classifying. While this would not stop the exponential cost, it would reduce the actual cost of each comparison. Some initial research in general recognition has shown that using a subset of features can be nearly as accurate as a full feature set (Blagojevic et al., 2010, Stahovich and Lin, 2016). However, this would still be an \(O(m^2)\) cost: it is unknown how much difference the reduction in \(m\) would make.

And some alternate programming approaches, like dynamic programming, may help to reduce the overall cost. This would work by reducing the number of comparisons that need to be made for each stroke.

For the recurrent sequence, it is important to note that these operations need to happen for every stroke added. This leads to a cost in not only the classification operation but also the grouping and anchoring operations. As grouping needs to consider other strokes in proximity, this may also result in an exponential cost increase. The same optimisation techniques may also work for grouping and it may be possible to combine the two operations (Wang and Raghupathy, 2007). As anchoring only needs to consider each annotation (a group of strokes) this cost will be less than the number of strokes.

In addition to the computational costs, the human interactions also need to be considered. People expect their annotations to remain static while they are writing them; although they are more flexible to adaptation afterwards. Therefore, it is critical that the strokes remain unchanged during annotating. While the
grouping and anchoring operations may potentially change information about
the annotation this should not be obvious to the user.

There is the question of whether users should know which strokes are grouped
together. Observations during the user studies indicate the participants were not
concerned about the bounds of an annotation; even when the bounds influenced
how they were grouped! As such, this may only be a technical issue and not
important to usability.

In contrast to the recurrent sequence, the one-off sequence executes a reduced
set of operations each time a stroke is added (see Figure 6.2). This sequence
assumes that once an annotation is added, its type and anchor cannot be changed,
only new strokes can be added to it. This reduces the cost in two ways. First,
there are fewer operations to perform for each stroke. Second, the cost of
classification is reduced as the stroke is either considered on its own or only with
information from previous strokes. This reduction comes at a cost of accuracy as
less information is available. As the studies show, this reduction in accuracy can
be very significant (see Chapter 4.)

Another challenge with the one-off sequence is the order of operations. In
Figure 6.2 this is shown as recognise then anchor. These two operations could
potentially be revised, or they could iterate. This is because information from
each step is needed to improve the accuracy of the other step – a chicken and
egg situation! I compensated for this in my studies by guiding the user to group
strokes using the post-it note metaphor.

When comparing the human costs, the one-off approach can be confusing for
users as people typically think of annotations as a whole based on their location,
rather than when the strokes were added. Because annotations can only be
added to, not changed, the implementation does not have to consider any effects
of changing annotations. The reduced accuracy can increase potential confusion
if any adaptation is needed in future.

This section discussed the two general sequences for the adding operations,
showing there are a number of trade-offs that must be considered when deciding
which sequence to use. The next section now discusses one of these operations:
recognition.
6.2.2 When is Recognition Needed?

An assumption underlying the second study (See Chapter 4) was that it is important to know the category of annotation. The findings in the study question this assumption. Technically, it is important to know the annotation type but often the participants did not care. From a technical viewpoint, knowing the annotation category is important for refitting (and other functionality) as each category has a different approach for refitting. If the wrong approach is used, then the annotation loses its meaning and can confuse the reader. However, to the participant, classifying the annotation was only important when the annotation was being refitted: if the content was unlikely to change then they did not want to classify the annotation. Thus, the real issue is when should annotations be classified.

An alternate approach is to only classify the annotation when needed. If the underlying context never changes then it does not matter what the annotation category is. If the context does change, then the system would attempt to automatically classify the annotation and allow the user to change it at this point. The user works together with the system to classify only the annotations that are changing; thus, reducing the workload on both the user and the system.

Concluding that 100% classification is not needed has important flow-on implications. One implication is how to involve the user. With the current implementation the user is only involved in classification when the annotation is added. With a classification-as-needed approach, the user needs to ensure the annotation is correctly classified when the context is changing. At the same time, if the user does not care about the annotations then they would not want to be interrupted to classify the annotation. This is another area that requires further investigation.

Another flow-on implication would be how to generate the anchors. The approach used in this thesis required knowing the annotation category; which is a separate concern from adapting the annotations. How the annotation is anchored affects how the annotation is repositioned. While repositioning was not investigated in this thesis, prior studies used the annotation category to anchor the annotations (Priest and Plimmer, 2006, Chen and Plimmer, 2007, Sutherland, 2012). There may need to be some trade-off between the recognition needed to generate a correct anchor. As it is unknown how important classification is for refitting, it may be necessary to do a basic classification for anchoring, and
a more precise classification for refitting. A future study could investigate the impact of not classifying prior to anchoring.

Another area that was not investigated was the importance of syntax for recognition. One important advantage of code is it is machine-readable: a compiler can extract syntactical information about the code without needing human input. However, based on the locations of the annotations observed in all three user studies, it is unknown whether understanding the underlying syntactical information would be of any benefit to refitting. Instead, knowing this information may be of more benefit for repositioning. Prior research with HTML documents suggests knowing the underlying HTML element can help with repositioning (Brush et al., 2001, Plimmer et al., 2010). Thus, another area for future study is: What impact would knowing the underlying syntax have on repositioning and refitting?

6.2.3 Changing the Appearance of Annotations

There has been very little prior research on refitting annotations (see Chapter 2). Prior research indicates annotations should refit to preserve the meaning of the annotation (Golovchinsky and Denoue, 2002, Bargeron and Moscovich, 2003). The meaning of annotations is related to not only their location relative to the text but the individual words within the text. As an example, people may underline two or three words in a sentence. It is these words that are significant: splitting the words or adding additional words in between may change the meaning of the annotation.

However, there is a caveat to refitting: an annotation should only refit if it preserves the meaning and it is needed (i.e. do not need to extend underlines beyond the length of the line). The research with Callisto suggests there is a cut-off point when annotations should not refit and the underlying context locked instead (Bargeron and Moscovich, 2003). However, the authors were unable to determine a common cut-off point across all their participants. In contrast, my work suggests, with a few exceptions, that annotations should always be refitted. The difference in results may be because a wider range of refitting algorithms was implemented (Callisto only implemented one per category of annotation), the tasks in this study were more constrained or the changes were less extreme. Alternatively, another reason for the difference may be the type of document investigated (code vs. news articles). As code is highly structured, programmers...
may expect annotations to conform to this structure. In contrast, the rules for a news article are looser with a wider variety of interpretations.

One important finding from this research is there were two, almost equally preferred approaches for refitting annotations. The first approach is to stretch the annotation so it maintains its original look and feel. The participants in this group preferred this style for all annotations: horizontal lines, vertical lines and enclosures. The second group preferred the annotation is split with some visualisation indicating the parts are a whole annotation. I have labelled these groups as ‘natural’ and ‘indicator’. This is something that has not been identified in the prior literature.

For the natural group, the annotation should be scaled so the start and end anchors maintain their locations and the annotation between retains its original look. For vertical lines and multiple line enclosures, this means scaling vertically. For horizontal lines, this means scaling horizontally: if the associated line splits across two lines then the annotation should split in the associated position but otherwise continue to scale. Scaling single line enclosures is unknown but I speculate they should refit in the same way as horizontal lines.

For the indicator group, the annotation should split where the changes to the underlying text are. This means if there are multiple changes there should be multiple splits. In addition, some visualisation should join the segments together. For vertical lines, a dashed vertical line connecting the segments was the most preferred visualisation. For horizontal lines, I only investigated adding perpendicular dashed lines, but a horizontal connecting line may also be preferred. For enclosures, it was more difficult to determine a preferred approach as sometimes a connecting line was preferred – other times perpendicular lines were. This appears to be related to the curvature of the enclosure at the breaks: as the enclosures became more vertical, the connecting lines were preferred. However, the approach must remain consistent, otherwise it is confusing to the user.

Finally, connectors were investigated as a separate group. The preferred approach was to keep the original look and feel of the annotation. The participants often preferred the unmodified approach as long as the associated note was still associated. These annotations are a good candidate for non-adaptation. I did not investigate any split approaches for connectors. The main reason for this was the difficulty in finding a split approach that still looked like a connector. Using
Chapter 6. Discussion

Figure 6.3: Design of Editor for Collecting Annotations. The top stroke will be unclassified while the bottom stroke will be classified as a connector.

the same split approaches as vertical lines tended to distort the annotation\(^2\). The results from the study indicate that either algorithm can be used as long as the original look is maintained.

6.2.4 Implementing Freeform Annotations

When designing an annotation system in an Integrated Development Environment there are two main considerations: how the users add the annotations, and what functionality will be supported? This section addresses these two considerations and suggests how annotations could be implemented.

For adding annotations, the research presented in this thesis suggests people think of annotations as static entities, with minimal need for adaptation. This implies the focus should be on ensuring the process of adding annotations is as simple and natural as possible. However, there are some annotation types that do need to be recognised for adaptation.

In terms of recognition, the user study in Chapter 4 suggests only two types of stroke need to be recognised when they are added: connectors and notes. This reduced set of annotations makes the classification process much simpler, especially when combined with a collaborative intelligence approach. Connectors were more difficult to correctly identify as they were often confused with single line annotations.

\(^2\)The original focus group disliked all of these algorithms for connectors.
Figure 6.3 shows the basic design of the code editor. The screen is split into two sections: the left side is normal while the right side has a semi-transparent overlay (blue in the design). Only two types of strokes are classified. Strokes that cross into or out of the blue region are classified as connectors: a new note is automatically started at the end point within the blue region. The other type of stroke classified, is a stroke completely within the blue region. This also automatically starts a new note, with the stroke as the first stroke in a test annotation. All other strokes will only be classified if the code underneath is changed.

This design will also handle grouping. The only annotations that need to be grouped are notes. Any strokes added within a note will be treated as a single annotation; any added outside will start a new annotation. If a stroke is close to the edge of the note, or crosses the edge, the note will resize to ensure there is adequate space. There is still an unanswered question as to whether notes should merge or not. Merging may be one way to handle if strokes were inadvertently added outside a note when they should have been inside it.

With these simple classification rules, it is easy to reposition annotations. Unclassified annotations are anchored to their starting location and will reposition relative to this location. Connectors are similar but use the left-most end point as their anchor. Notes use the top-left corner of the original note box.

The second main consideration is what functionality annotations will support. One key point identified in the literature review is freeform digital ink annotations have the potential to transcend what is available on paper. Thus, by reviewing some of the limitations of paper we can build a better idea of what may be beneficial. Two such limitations are refinding and sharing annotations (Price et al., 1998a, Schilit et al., 1998a, Steimle, 2009, Steimle et al., 2009).

Refinding annotations involves returning to an existing annotation at some point in the future. This is often associated with an activity like “where did I see...?” or “what did I think when I read...?” Both of these activities are about refreshing a person’s memory of what they were reading and engaged with at a point in time. One of the benefits of active reading is that the deeper engagement allows the reader to comprehend more of what is read (Adler and Van Doren, 1972). But at times the reader needs to move on and so records information on the page so they will not forget it later. The reader then either relies on where they remember writing the annotation (a form of spatial memory) or flicks through the pages to find it (a form of visual scanning). Both of these methods have
their limitations and require time to apply. With computer systems, it is easy to index annotations and display them to the reader. The reader could then scan through the list of annotations directly, thus reducing the time needed to find the annotations and their associated context and information (Chen and Plimmer, 2007).

In contrast to refinding annotations, sharing annotations is a collaborative process. One reason for adding annotations is for sharing information with other people (see §3.3.3). Potentially, sharing annotations has two forms: online and offline. With online sharing, the other person can immediately view the annotations added by the other person. Traditionally online annotating is done with the same document and has benefits in certain situations; like teaching and reviewing. However, there are space constraints with using paper as all participants need to at least see the paper and what the reader is annotating. In contrast, offline sharing assumes a period of time between the reader annotating and someone else reading the annotations. Offline annotation is a form of communication by annotation, that evolved over time as a mechanism for scholars to communicate with each other (Agosti et al., 2007); thus, it is a form of annotating that people are familiar with.

Previous studies have suggested a number of ways to refind annotations. In Integrated Development Environment studies annotations can be added to the visual interface (Chen and Plimmer, 2007, Sutherland and Plimmer, 2013). One approach is to show the annotations in a tool window with either a list or a tree structure. One important finding from these studies is the context is important: annotations by themselves have less value (Sutherland, 2012). Other forms of navigation include a “reader’s list” (Price et al., 1998b, Schilit et al., 1998a), margin links to other information/annotations (Price et al., 1998b, Schilit et al., 1998a), searching (Shipman et al., 2003), bookmarks (Chen et al., 2012, 2013), code maps (Lichtschlag et al., 2014), tool tips (Sutherland, 2012) and timelines (Cattelan et al., 2008).

A second aspect of refinding that is relevant to code-based annotations, is which document has annotations. Most implementations only displayed the annotations for the current document (Price et al., 1998b, Schilit et al., 1998a, Cattelan et al., 2008, Chen et al., 2012, 2013, Sutherland and Plimmer, 2013). In an Integrated Development Environment the code could be split across multiple documents; making it important to not only know where the annotation is but which document it is in.
Comments from the participants in the user studies also provide insight into how refinding could work. Several participants in the studies mentioned they would like some way to search their annotations. This could either be based on the content of the annotation ("can I search for the annotations where I wrote TODO?") or the metadata for the annotation ("can I search for annotations that were added last week?") Other participants mentioned they would like some way of moving from annotation to annotation ("I know I marked similar things the same way; can I use these as links?") Finally, another commonly mentioned approach was to have a common diagram of the code with links to specific locations. This appears to be similar to the functionality provided in CodeGraffeti (Lichtschlag et al., 2014). At the same time, it was important for the participants that the navigation functionality provided not add to their workload. Several participants mentioned they would only use it if it worked in the background.

Based on these aspects, an implementation would have both a local finder for within a document and a global finder across multiple documents. If annotations are displayed in a separate toolwindow, they should display both the annotation and its context (Chen and Plimmer, 2007, Sutherland and Plimmer, 2013).

The global display could display all the annotations added to the project (Chen and Plimmer, 2007). The user should be able to reorder the list (e.g. by time added, person added or location) and search. Ideally, this would be some form of "smart" search: give me all the annotations I added last week related to method definitions (Shipman et al., 2003). While it would be nice to also search based on the annotation content (e.g. any text) this would require better recognition than is currently available.

There are a number of potential options for within-document navigation. First, a separate toolwindow could show all the annotations for the document (Chen and Plimmer, 2007). This would function as a filtered version of the global display with only annotations for that document. One important difference from prior versions (Chen and Plimmer, 2007, Sutherland and Plimmer, 2013) is it must display the context (see Figure 6.4). A sidebar could be added showing where there are annotations within the document. Hovering over an indicator would show a tooltip of the annotation and its context, clicking on the indicator would navigate to the annotation and ensure it is visible. Finally, next/previous indicators could be added showing when there are more annotations above or below the currently displayed code. A potential design is shown in Figure 6.4.
Figure 6.4: Proposed within-document navigation elements. Tool-window is to the right, the pop-up shows a selected annotation and the blue button to the centre of the bottom is the indicator to the next annotation.
For sharing, the functionality would be based on when the annotations are shared. Offline sharing would work in a similar way to the person adding annotations themselves. The user interface could be enhanced by storing metadata on who added the annotations and a list of recently added annotations by other people. One consideration with offline sharing is whether the people would want to share annotations (Steimle, 2012); however, as Marshall (1997) documented, even annotations that are unintentionally shared can benefit other readers.

In contrast, online sharing would act as a communications interface. All annotations would be shared instantaneously between connected implementations. There are two main challenges for this form: how should navigation be handled (Lichtschlag and Borchers, 2010) and how should users be notified of new annotations (Wu et al., 2008)?

This section discussed how annotations could be implementing an Integrated Development Environment in a way that flows with the user’s natural annotation style. It includes what should be included and what can be omitted. This section concludes the main body of the discussion; the next section summarises the key points from this chapter.

### 6.3 Summary

This chapter tied together the different threads of the research in this thesis and discussed how it fits within the broader context.

The first section focused on how and why programmers annotate currently and what implications this has for implementing a freeform digital ink annotation system.

The second section focused on the technical issues of implementing freeform digital ink annotations.

The next, and final, chapter now concludes the thesis and suggests future directions of study for freeform digital ink annotations.
Conclusions
The Day the Pen Stood Still

“Anybody who has been seriously engaged in scientific work of any kind realizes that over the entrance to the gates of the temple of science are written the words: ‘Ye must have faith.’”

Max Planck

In this thesis, I present my investigations into freeform digital ink annotations. While the focus of the research in this thesis is freeform digital ink annotations in an Integrated Development Environment, the results are applicable to other areas of freeform digital ink annotations on dynamic digital documents.

This final chapter outlines the key conclusions of my research. Section 7.1 reviews the research questions. This is followed in section 7.2 by the list of key contributions from this research, both for code comprehension and dynamic freeform annotations. Then, section 7.3 outlines some directions to extend this research in future. The final remarks are in section 7.4.

7.1 Research Questions Revisited

In this thesis, the main research objective was “how we can support freeform digital ink annotations for programming in an Integrated Development Environment?” Two questions were asked to guide the investigation.
The study in Chapter 3 investigated RQ1, “How and why do programmers annotate programming code on paper?” The results indicate that experienced programmers annotate on printed code in a similar manner to annotating other forms of text. The three main reasons for annotating were: to assist with navigation, to offload information for later use, and to record information for others. The findings suggest that freeform annotations in an Integrated Development Environment may be a useful tool for code comprehension.

The studies in Chapters 4 and 5 investigated RQ2. “How can an Integrated Development Environment be extended to support annotation of digital dynamic documents in a natural way?” Each study examined a different facet of this question.

The study in Chapter 4 explored “How can the recognition of annotations be improved?” (RQ2.1). The results show that using a collaborative intelligence approach can improve recognition. However, an unanticipated finding from this study was the participants did not expect 100% accuracy for recognition. Instead, the results indicate that only some annotations need to be recognised.

Finally, the study in Chapter 5 explored “How can annotations be automatically refitted in response to changes in the code?” (RQ2.2). The results show that knowing the category of annotation is important for refitting annotations as each category needs a different approach. Surprisingly, two different styles were preferred by the participants: either maintaining the natural look of the annotation or splitting the annotation and adding a visualisation. These preferences were consistent, with each participant preferring the same style across all annotation categories.

Given these results for the research questions, the next section outlines the key contributions from this thesis.

### 7.2 Contributions

This thesis presents the following key contributions:

1. A systematic literature review, outlining the current state of freeform digital ink annotations, which includes the current state of research (published in Sutherland et al., 2016a).
2. An investigation into if and how programmers would annotate program code. This includes details on the characteristics of their annotations and some insight into the reasons why they annotate (published in Sutherland et al., 2015).

3. An implementation of an extensible tool that allows exploring different facets of freeform digital ink annotations in a programming environment.

4. An investigation into alternate input methods for improving the recognition accuracy of single strokes. This revealed how even simple mechanisms, such as constraints and verification, can increase accuracy, but at a cost of increased work for the user.

5. An initial set of proposals for how annotations should refit, based on the category of annotation and preferences of the user (published in Sutherland et al., 2016b).

7.3 Future Work

This thesis provides an introductory exploration into freeform digital ink annotations in an Integrated Development Environment. While this provides some details on how people expect annotations to behave, the results introduce several associated questions.

The initial question focused on how annotations could assist programmers in a program code comprehension task. The initial user study showed programmers would still annotate if possible and indicated navigation and information offloading as important areas. This study was with experienced programmers; those who were already familiar with reading and understanding program code. Novice programmers may have a different set of reasons for annotating. Studying this group would then guide how to build tools that benefit a wider range of programmers.

In addition, the participants used only used pen and paper: annotating within an Integrated Development Environment might produce different annotations. One reason for the difference is an Integrated Development Environment already provides a rich set of functionality that is not available on paper. Rather than duplicating the functionality, annotations should aim to provide new functionality that extends what is currently available. Freeform annotations may provide
a lower mental overhead than existing tools, thus making it easier to explore and bookmark the code. In addition, the lower mental overhead may allow programmers to jot notes without losing focus on what they are reading.

Another question is how to build a tool that extends Integrated Development Environment functionality without interfering with the existing functionality. This needs to be built in a way that does not increase the mental workload of the reader as this may cancel the benefits of annotations.

In addition to the domain of understanding program code, freeform digital ink annotations may be beneficial in other domains. There are additional tasks related to working with code. Another related programming task is debugging code. While there are already debugging tools available (e.g. Bragdon et al., 2010, DeLine et al., 2012), annotations may allow offloading of information during the debugging process, which in turn would allow for increased working memory to help with debugging.

Freeform annotations may also be useful for documenting code. Several participants mentioned the ability to draw diagrams associated with the code would be valuable. However, previous research indicates this is not as easy as it appears (Lichtschlag et al., 2014). In order for the documentation to be relevant, it must somehow be associated with the code. This could either be a separate window (as implemented in CodeGraffitt) or linked via connectors (as I explored). Either way, these anchors would need to correctly adapt as the code changes. The diagrams would also need to be stored in a similar manner to code, preferably in a version control system. Further work is needed on how to store annotations in this manner.

Related to documentation are code reviews and versioning. Code reviews are often performed after the code has been changed as it is important to know where the code has changed. During code reviews, the reviewer potentially needs to know what annotations were previously associated with the changed code.

Code review is not the only form of collaboration. One rich area of investigation is lecturing and teaching. Researchers have already found that freeform ink can assist in lectures (Anderson et al., 2004a, 2005). This could be extended by looking at how annotations in an Integrated Development Environment could help. Potentially a lecturer could receive code from students or push code to
students for them to review and annotate. Having a mechanism to control these interactions may assist learning.

Most of these directions have focused on the usage of freeform digital ink annotations. In addition, there are still a number of outstanding technical challenges for freeform ink. First, these studies focused on refitting annotations when the underlying text changes. To constrain the problem, the focus was on adding additional text. To date, there has been no work on how annotations should refit when text is deleted underneath part of an annotation. Additional work is also needed on how single line enclosures should refit. And there are potentially other categories of annotations that could be refitted (in the current research these are grouped together under complex annotations) (Bargeron and Moscovich, 2003).

In addition to refitting, there is also repositioning and orphaning. Repositioning has been investigated and has robust solutions for freeform annotations. In contrast, orphaning has not. There are solutions for text-based annotations but it is unknown whether the same approaches would apply for freeform annotations.

Finally, work is still needed on the adding operations (recognising, grouping, anchoring and storing). While there are usable solutions for each of the operations, these could all be improved. For example, Chapter 4 identifies how some simple heuristic rules provide a similar accuracy to a classifier built in RATA (Blagojevic et al., 2010). These rules could be refined with additional features or machine learning algorithms.

### 7.4 Conclusions

This thesis has explored supporting freeform digital ink annotations for programming in an Integrated Development Environment. Despite its exploratory nature, this thesis offers some insight into both the value of freeform annotations and how they can be supported.

In terms of value provided, the results indicate that freeform annotations assist with navigation and information offloading. It would be interesting to assess how freeform annotations in an Integrated Development Environment can assist with these two areas, especially without conflicting with existing functionality.
For supporting annotations, the results show collaborative intelligence can improve the classification rates when compared to an automatic approach, and annotations should be refitted. In addition, there are two generally preferred styles of refitting. One unexpected result from the recognition study is the participants did not expect all their annotations to be correctly classified.
Ethics approval letters

This appendix includes the ethics approval letters for the studies included in this thesis.

The first letter, received from UAHPEC on 25th May, 2014, is for the study investigating how experienced programmers annotate program code on paper. This study was described in Chapter 3.

The second letter, received from UAHPEC on 8th March, 2015, is for the studies in Chapter 4 and Chapter 5. These two studies were performed concurrently under the same ethics approval.
MEMORANDUM TO:

Dr Beryl Plimmer
Computer Science

Re: Application for Ethics Approval (Our Ref. 011501): Approved

The Committee considered your application for ethics approval for your project entitled **Investigation of computer program code annotation on paper**.

We are pleased to inform you that ethics approval is granted for a period of three years.

The expiry date for this approval is 25-May-2017.

If the project changes significantly, you are required to submit a new application to UAHPEC for further consideration.

If you have obtained funding other than from UniServices, send a copy of this approval letter to the Research Office, at ro-awards@auckland.ac.nz. For UniServices contracts, send a copy of the approval letter to the Contract Manager, UniServices.

In order that an up-to-date record can be maintained, you are requested to notify UAHPEC once your project is completed.

The Chair and the members of UAHPEC would be happy to discuss general matters relating to ethics approvals. If you wish to do so, please contact the UAHPEC Ethics Administrators at ro-ethics@auckland.ac.nz in the first instance.

Please quote reference number: **011501** on all communication with the UAHPEC regarding this application.

(This is a computer generated letter. No signature required.)

UAHPEC Administrators
University of Auckland Human Participants Ethics Committee

c.c. Head of Department / School, Computer Science
Dr Andrew Luxton-Reilly
Mr Craig Sutherland
MEMORANDUM TO:

Dr Beryl Plimmer
Computer Science

Re: Application for Ethics Approval (Our Ref. 012453): Approved with comment

The Committee considered your application for ethics approval for your project entitled Investigation into digital ink annotation refitting for program code.

Ethics approval was given for a period of three years with the following comment(s):

1. Please ensure that the UAHPEC approval wording shows the current approval date.

The expiry date for this approval is 08-Mar-2018.

If the project changes significantly you are required to resubmit a new application to UAHPEC for further consideration.

In order that an up-to-date record can be maintained, you are requested to notify UAHPEC once your project is completed.

The Chair and the members of UAHPEC would be happy to discuss general matters relating to ethics approvals if you wish to do so. Contact should be made through the UAHPEC Ethics Administrators at ro-ethics@auckland.ac.nz in the first instance.

All communication with the UAHPEC regarding this application should include this reference number: 012453.

(This is a computer generated letter. No signature required.)

Secretary
University of Auckland Human Participants Ethics Committee

c.c. Head of Department / School, Computer Science
Mr Craig Sutherland
Dr Andrew Luxton-Reilly
**vsInk Architecture**

**vsInk** is a framework for collecting freeform digital ink in the *Visual Studio* code editor. This appendix describes the implementation details of **vsInk**. **vsInk** was originally developed as an honours project (Sutherland, 2012). The details in this appendix describe the modified version of **vsInk** developed for this thesis. As the version used for this thesis defers from the original version, the reader is referred to Sutherland (2012) and Sutherland and Plimmer (2013) for the original implementation details.

B.1 describes the basic architecture of **vsInk**, including the internal data structure, main UI layers and pipelines for adding and rendering annotations. B.2 describes the mechanism for segmenting and anchoring annotations as they are added. B.3 describes the pipeline for rendering annotations. B.4 describes adornments, how they are generated and the default adornments in **vsInk**.

This chapter refers to a number of interface definitions. These definitions are detailed in Appendix C.

### B.1 Basic Architecture

Figure B.1 shows **vsInk** in *inking* mode. **vsInk** extends the standard code editor in *Visual Studio*. By default, **vsInk** is disabled: when **vsInk** is enabled a text icon is added to the bottom right-hand corner of the code editor. Any mouse, touch or pen input is now treated as ink input for generating annotations. The only external UI element is a small toolbar that allows the user to enable/disable
vsInk and to change between the different modes (drawing, highlighting and erasing\(^1\).)

The following sub-sections describe the internal data structures, the UI layers, and the two pipelines: adding strokes and rendering annotations.

### B.1.1 Data model

The annotations are stored as an array of `Annotation` instances, with one array per file. Each `Annotation` instance consists of a set of metadata about the annotation and an array of `AnnotationSegment` instances. Each `AnnotationSegment` is associated with one or more ways of displaying the annotation. Each `AnnotationSegment` has metadata plus two arrays: Stroke instances and Anchor instances. The Stroke instances contain the associated WPF ink data. Each Anchor instance contains a tracking point for the code and an offset from the top-left corner to the start point of the segment. All strokes are based relative to this start point.

Each `AnnotationSegment` provides the complete visual and anchoring details for a segment of the annotation. This allows each segment to be rendered

\(^1\)There are also two additional modes for usability testing: selecting text and selecting ink. These are not intended for normal use by a user.
**B.1.2 UI Layers**

`vsInk` adds four additional layers to the code editor (see Figure B.2.) These layers stack on top of the default layers of *Visual Studio*.

The bottom layer, *Background*, provides UI elements that are generic to *vsInk*. These are elements that do not change with the annotations. An example is the blue region used in the collaborative intelligence recognition (see Figure 4.5.) independently of the other segments. Each *Annotation* must contain a segment of type *Primary*: this maps to the original, unmodified annotation. In addition, it may contain one or more *AnnotationSegment* instances of type *SegmentedRough* or *SegmentedFine*. These provide two levels of granularity for segmenting annotations. These are explained in B.2 below.
FIGURE B.3: Pipeline for processing digital ink. This shows the operations needed to convert from an ink stroke into an annotation that can be rendered on a document.

The next layer, Underneath, is used to display annotation adornments. Adornments are described in more detail in B.3 below.

The following layer, InkCanvas, is where the ink collection occurs. This layer records the digital ink added and notifies the adding pipeline. Because this layer is interactive, the only elements added to it are the rendered strokes.

The final layer, Foreground, is used to position UI elements that the user can interact with for an annotation. An example of an element on this layer is the verification drop-down (see Figure 4.4.)

### B.1.3 Processing Digital Ink

Internally vsInk uses a pipeline for processing incoming digital ink (see Figure B.3.)

The adding pipeline is triggered when the user draws an ink stroke in the editor. InkCanvas fires an event notifying vsInk that an ink stroke has been collected. vsInk then records the basic details about the stroke and passes these details to a stroke recogniser. The recogniser is responsible for classifying the stroke. A new stroke recogniser can be added to vsInk by implementing IRecogniser interface. At any time, there is one and only one recogniser active: the currently active recogniser is configured using the options dialogue in Visual Studio.

After the stroke has been recognised, vsInk attempts to match it with an existing annotation that is visible. The purpose of this match is to determine whether the stroke can be merged with an existing annotation. The merge detection is performed by the IStrokeTypeProcessor instance of the existing annotation. If the stroke can be merged it is added to a list of merge possibilities. Once
all the annotations have been checked, the annotation with the highest merge ranking\(^2\) is selected as the merge target. If there were no merge possibilities, a new annotation will be started.

To start a new annotation, the stroke is passed to an \texttt{IStrokeTypeProcessor} instance. Which \texttt{IStrokeTypeProcessor} is used depends on the stroke type of the new stroke. The \texttt{IStrokeTypeProcessor} is responsible for generating the new annotation, including segmenting and anchoring. Further details on this step are included in B.2. The new stroke is then added to the list of annotations. A new \texttt{IStrokeTypeProcessor} implementation can be added to \texttt{vsInk} to change how new annotations are generated or when a new annotation category is added.

There are two different approaches for merging annotations. Both approaches use an instance of \texttt{IMergableStrokeTypeProcessor}. In the first approach, the new stroke is added to an existing annotation. The stroke is repositioned so it matches the anchors from the original stroke. The second approach adds a new annotation and links the new annotation to the existing one. Both annotations have their own anchors: these anchors are independent of the other annotation.

Once the stroke has been processed by either an \texttt{IStrokeTypeProcessor} or an \texttt{IMergableStrokeTypeProcessor}, the document is marked as dirty in \texttt{Visual Studio}. Saving the annotations is performed when the user saves the associated code document. Saving the annotations writes the details of the annotations to a file in the same folder as the code file. The name of the file is the same as the code file but with the extension ‘.vsInk’. The annotations are stored in a proprietary JSON format.

\(^2\)Currently the only merge ranking is the order the annotation was added.
B.1.4 Rendering annotations

A second pipeline is used to render annotations in response to changes in the editor (see Figure B.4.)

The first step is to remove all elements currently on the inkCanvas and adornment layers, as the elements produced by the renderers and adornment generators are not tracked.

The second step depends on the renderer. Each renderer either renders the entire annotation or the segments of the annotation. Both approaches work using the same general principals. The main difference is the number of times the renderer is called. If it renders the whole annotation, it is called only once. Otherwise, it is called once per segment.

First, the anchor location of the annotation or segment is calculated using the tracking point and offset. This calculation uses the same rules as the original vsInk (Sutherland and Plimmer, 2013). If the anchor is currently not displayed the renderer will stop, thus leaving the annotation or segment unrendered (and not visible). Next, the renderer generates a copy of all the strokes in either the annotation or segment. These strokes are translated to the correct position on the InkCanvas layer. Any additional strokes are then generated (e.g. joining lines). Finally, all the strokes are returned from the renderer.

Once all the renderers have completed, the strokes are added to the InkCanvas layer.

The final step of the pipeline is to generate and add the adornments. This is a two-phase process. First, adornments are generated for each segment of an annotation, then the adornments for the entire annotation. Once all the adornments have been generated they are added to the Adornment layer.

B.2 Segmenting and Anchoring Annotations

Anchoring and segmenting are important pre-requisites for rendering annotations. These two operations are performed within the same step of the pipeline, within either the Start new annotation or the Merge with Existing steps of the adding pipeline.
TABLE B.1: Segment and anchor data for each annotation type.

<table>
<thead>
<tr>
<th>Annotation Type</th>
<th>Segments</th>
<th>Anchors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single line</td>
<td>Whole</td>
<td>First and last characters</td>
</tr>
<tr>
<td></td>
<td>By word</td>
<td>First character of each word</td>
</tr>
<tr>
<td></td>
<td>By Character</td>
<td>Each character</td>
</tr>
<tr>
<td>Multiple line</td>
<td>Whole</td>
<td>First character of top and bottom lines</td>
</tr>
<tr>
<td></td>
<td>By line</td>
<td>First character of each line</td>
</tr>
<tr>
<td>Connector</td>
<td>Whole</td>
<td>First character of top and bottom lines</td>
</tr>
<tr>
<td>Complex</td>
<td>Whole</td>
<td>First character of top line</td>
</tr>
</tbody>
</table>

The segmentation and anchoring processes are dependent on each annotation type. Table B.1 shows how each annotation is segmented and anchored.

Segmentation is performed first. The stroke processor segments the annotation into characters (for single line annotations) or lines (for multiple line annotations.) The result of segmentation is a new set of ink strokes grouped into segments. These segments are concatenated into a single array, together a single segment for the whole annotation. Anchors are then generated for each segment.

Segmentation may also add new points to a stroke. Each stroke can be decomposed into a series of lines, with some of these lines crossing a segment boundary. When this occurs, an intermediary point is calculated for where the line would cross the segment boundary. This is calculated by extrapolating a straight line between the points and calculating the new point using Euclidean geometry.

The first step in anchoring is to determine the relative lines in the underlying code. Visual Studio provides an API that converts a point on the screen to a line and character position. As Visual Studio does not use absolute screen coordinates the absolute coordinates are converted into Visual Studio relative coordinates and then the API is called.

Once the line has been determined, vsInk calculates an offset from the first non-whitespace character in the line to the base point in the segment. This base point is determined by the processor: it is typically the top-left corner of the bounding box or the first point in the stroke. The entire segment is then translated relative to the base point to simplify rendering.

Each segment has at least a primary anchor, some may also have secondary anchors. Having multiple anchors is needed for some of the rendering algorithms (e.g. stretching).
The result of segmenting and anchoring is a set of segments for an annotation. These segments are stored together within the annotation. Each segment contains the translated strokes, together with any generated points, and one or more anchors.

### B.3 Rendering Annotations

Annotations can be rendered in one of two ways: *by segment* or *by line*. The approach defines the order in which segments are sent to the `StrokeRendererBase` instance. In the *By Segment* approach, each segment is rendered in the order that it was originally added. The current segment, together with the next and previous segments are passed to the renderer as these are needed for some adaptations. The *By Line* approach groups the segments by underlying text line and then renders all the segments in the line, from top to bottom. The renderer receives only the segments for a line.

The actual rendering of an annotation is performed by a `StrokeRendererBase` instance. The following steps detail the general process that is followed by the renderers:

1. **Calculate the position on the screen.** The position is calculated by retrieving the anchor position of the segment in the code and converting the position to screen coordinates. The offset is then added to these coordinates, providing the base position for the segment. The segment is then translated relative to these coordinates. As the annotation is stored so the base point is at (0,0) this moves the entire annotation into the correct position relative to the anchor.

2. **Refit the annotation.** This step is specific to each renderer; however, there are three general approaches: *leave unmodified*, *scale*, or *add lines*. *Leave unmodified* just copies the strokes in the segment. *Scale* generates a new copy of the stroke and applies the scale factor relative to the segment’s base point. *Add lines* copies the strokes in the segment and generates any additional lines as needed.

The renderer then returns the set of strokes to `vsInk`. These are collected while the renderers are running and added to the `InkLayer` when all the renderers

---

3 Whole segments are treated as *by segment*. 
have finished. Any general UI elements are generated at this point. Finally, the rendered annotations and segments are passed to the adornment generators.

### B.4 Adornments

Adornments allow the display extra information about an annotation and/or its segments. The adornments are standard WPF elements added to the *Adornment* layer in *vsInk*. There are only three limitations on what is possible:

1. Adornments are non-interactive. These are rendered along with the annotation and cannot either be changed or used by the user.
2. Adornments are associated with an annotation or segment. If the annotation or segment is not displayed then the adornment will not be either.
3. The adornment cannot modify the annotation or segment; it can only use existing or calculated information.

Adornments are added by sub-classes of the *AdornmentGeneratorBase* abstract class. This class provides two methods for generating adornments: for the entire annotation or for a segment within an annotation. When the adornments are generated, *vsInk* will generate segment adornments first and then annotation adornments. By default, the elements are visually stacked in the order they are generated but it is possible to set the visual display order by specifying a z-index for the generator.

All adornments are standard WPF elements (*FrameworkElement* instances) as this provides flexibility for any type of adornment. However, this requires the adorner to correctly position the elements. The adorner has two approaches for positioning elements on the screen: manually positioning or vertical stacking. With manually positioning, it is the responsibility of the adorner to correctly position the elements. Vertical stacking will automatically handle the positioning and stack the elements in the order they were generated. This stack is then positioned to the right of the annotation. This approach is typically used for text but any UI elements can be added to the stack.

*vsInk* provides a number of inbuilt adornments. These adornments provide information about the annotation or an annotation’s segments. These adornments and a brief description are listed in Table B.2.
### Table B.2: Built-in adornments in vsInk.

<table>
<thead>
<tr>
<th>Adornment Name</th>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anchor Lines</td>
<td>Segment</td>
<td>Draws a line between the calculated anchors for each segment.</td>
</tr>
<tr>
<td>Anchors</td>
<td>Segment</td>
<td>Displays the location of the calculated anchor for each segment.</td>
</tr>
<tr>
<td>Annotation Centre</td>
<td>Annotation</td>
<td>Draws a red circle at the location of the centre of the annotation.</td>
</tr>
<tr>
<td>Annotation Number</td>
<td>Annotation</td>
<td>Displays the number of the annotation within the file. This number is set when the annotation is first added and never changed.</td>
</tr>
<tr>
<td>Anchor to Annotation</td>
<td>Segment</td>
<td>Draws a line from the base point to the first point in the segment.</td>
</tr>
<tr>
<td>Annotation Type</td>
<td>Annotation</td>
<td>Displays the type of the whole annotation (e.g. single line, multiple line, connector or complex).</td>
</tr>
<tr>
<td>Context Bounds</td>
<td>Annotation</td>
<td>Displays a bounding box around all the underlying lines of code. This is useful for seeing what overlap there is between the code and an annotation. This will generate one rectangle for the entire annotation.</td>
</tr>
<tr>
<td>Line Context Bounds</td>
<td>Annotation</td>
<td>Displays a bounding box for each underlying line of code. This is useful for seeing what overlap there is between the code and an annotation. This will generate multiple rectangles: one for each line of code.</td>
</tr>
<tr>
<td>Merge Bounds</td>
<td>Annotation</td>
<td>Draws a yellow note visualisation around an annotation. Any strokes added within this visualisation will be merged with the annotation. This adornment is only displayed for complex annotations.</td>
</tr>
</tbody>
</table>
### Table B.2: (continued).

<table>
<thead>
<tr>
<th>Adornment Name</th>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mid-line</td>
<td>Annotation</td>
<td>Draws a line that divides the annotation vertically in half.</td>
</tr>
<tr>
<td>Segment Height</td>
<td>Segment</td>
<td>Displays the height of the segment. This is displayed to the right of the annotation with a brace showing the top and bottom boundaries.</td>
</tr>
<tr>
<td>Segment Order</td>
<td>Segment</td>
<td>Displays the order number of each segment in an annotation. The order number is the order the segment was generated when the annotation was segmented.</td>
</tr>
<tr>
<td>Start-&gt;End Lines</td>
<td>Segment</td>
<td>Draws a line between the start and end points of each stroke in the annotation. This is useful for visually seeing the slope of the stroke.</td>
</tr>
<tr>
<td>Stroke Bounds</td>
<td>Segment</td>
<td>Displays the bounding box for each stroke within the segment.</td>
</tr>
<tr>
<td>Stroke Type</td>
<td>Annotation</td>
<td>Displays the original stroke type from the recogniser. The actual value depends on which recogniser is being used.</td>
</tr>
<tr>
<td>Tracking Point Brace</td>
<td>Segment</td>
<td>Generates a line between the top and bottom anchors for a segment (only for segments that have both anchors).</td>
</tr>
<tr>
<td>Tracking Point to Anchor</td>
<td>Segment</td>
<td>Displays a link between the tracking point (anchor in the code) to the base point of the annotation. This is the offset of the annotation.</td>
</tr>
</tbody>
</table>
Interfaces for Extending *vsInk*

*vsInk* was designed as a framework for future research. It contains a number of extension points in its processing pipelines (see Appendix B). This appendix provides the technical documentation on the interfaces for the extension points.

The following interfaces are documented in this appendix:

- **IRecogniser** defines a recogniser for classifying new strokes.
- **IStrokeTypeProcessor** defines a processor for generating new single stroke annotations.
- **IMergableStrokeTypeProcessor** defines a processor for generating new multiple stroke annotations and adding addition strokes to existing annotations.
- **StrokeRendererBase** defines a renderer for displaying (rendering) annotations in the code editor.
- **AdornmentGeneratorBase** defines a generator for adding adornments to annotations.

### C.1 IRecogniser

**IRecogniser** allows defining a new recogniser for classifying strokes. *vsInk* implements the one-off approach for recognising strokes (see Chapter 2): therefore this interfaces only processes a single stroke.

The following members need to be defined in a class that implements **IRecogniser**:

```c
void Activate();
```
void Initialise(string modelToLoad);

RecognitionResult Recognise(Stroke stroke,
                             Stroke previousStroke,
                             CodeFeatureSet codeFeatures);

Initialise() is called when the recogniser is first initialised. This method will only be called once for an instance of Visual Studio.

The following arguments are passed to Initialise():

- modelToLoad is the name of a model file; currently, this is hard-coded within vsInk to the file containing the RATA model.

Activate() is called when the recogniser is activated. This is used to perform any preparation actions before the first stroke is passed to the recogniser. If the user changes the recogniser (via the UI) then this method will be called prior to the next stroke being recognised.

There are no arguments for Activate().

Recognise() is called once for every stroke that is added to vsInk. This method takes in information about the current strokes and returns the classification of the stroke.

The following arguments are passed to Recognise():

- stroke is the stroke currently being processed.
- previousStroke is the previous stroke that was processed.
- codeFeatures are the context-specific features for the current stroke (see §4.3.1).

This method returns a RecognitionResult instance. This instance contains the Type of stroke and an optional set of Values. Values is used in the diagnostics interfaces. They can be used to display information about why the recogniser returns the specified Type.

The Recogniser attribute must be added to the class in order for vsInk to load it. The required argument for this attribute is the display name of the recogniser (this is the name that will be displayed in the UI).

If an IRecogniser instance introduces new stroke types, then an IStrokeTypeProcessor should also be added. Currently, the following stroke types are ‘known’ in vsInk:
Appendix C. Interfaces for Extending vsInk

- Brace
- Connector
- Enclosure
- Highlight
- MarginBar
- Symbol
- Text
- Underline

These types map to the current `IStrokeTypeProcessor` implementations.

C.2 IStrokeTypeProcessor

`IStrokeTypeProcessor` allows defining a new processor for generating new annotations from strokes. `vsInk` has one instance of this interface per stroke type.

The following members need to be defined in a class that implements `IStrokeTypeProcessor`:

```csharp
bool CanMerge(Annotation annotation,
             string strokeType,
             Stroke stroke);

IEnumerable<Annotation> Generate(Stroke stroke,
                                 IWpfTextView textView,
                                 GlobalSettings settings);
```

Prior to generating a new annotation, `CanMerge()` is called to determine whether a new stroke can merge with an existing annotation. If this method returns `false` then `Generate()` is called. Otherwise, the instance is cast as an `IMergableStrokeTypeProcessor` and `Merge()` is called (see C.3 below).

`CanMerge()` is called multiple times when a new stroke is added to a document. `vsInk` calls this method for every annotation that is currently visible. If an annotation does not accept new strokes (i.e. is not support merging) then this method should return `false`.

The following arguments are passed to `CanMerge()`:

- `annotation` is the existing annotation to check.
- `strokeType` is the type of the new stroke.
Appendix C. Interfaces for Extending vsInk

- **stroke** is the new stroke to check.

Generate() is if vsInk cannot find an annotation to merge the new stroke with. This method is responsible for generating a new Annotation instance. The generation includes segmenting and anchoring the annotation. The following arguments are passed to Generate():

  - **stroke** is the initial stroke of the new annotation.
  - **textView** is the Visual Studio code editor that the annotation is being added to.
  - **settings** are the global settings currently active in vsInk.

When this method is finished, vsInk will add the new Annotation instance to the list of annotations for the document and render all the annotations currently visible.

The StrokeTypeProcessor attribute must be added to the class in order for vsInk to load it. The required arguments for this attribute are the class of annotations it handles and a list of annotation types to handle.

### C.3 IMergableStrokeTypeProcessor

IMergableStrokeTypeProcessor extends IStrokeTypeProcessor and allows merging new strokes into existing annotations. The methods on an implementation of this interface are only called if the CanMerge() method returns true.

The following members need to be defined in a class that implements IMergableStrokeTypeProcessor:

**Thickness? GetMergeBounds(Annotation annotation);**

**MergeType GetMergeType(string strokeType);**

**void Link(Annotation annotation,**
  **Annotation newAnnotation);**

**ITextUndoPrimitive Merge(Annotation annotation,**
  **Stroke stroke,**
  ** IWpfTextView textView,**
  ** IRefreshable source);**
void ReAnchor(Annotation annotation,
            IWpfTextView textView,
            Anchor anchor,
            AnchorType type);

If CanMerge() returns true, then GetMergeType() is called to determine how the new stroke should be merged. If MergeType.Standard is returned, then Merge() is called to add the new stroke to an existing annotation. If MergeType.StandAlone is returned, then a new Annotation instance is generated via Generate() and then Link() is called to join the two annotations together. These methods are called once per new stroke.

In contrast, GetMergeBounds() is called as needed by vsInk to determine the merge bounds of an annotation. The merge bounds of an annotation are calculated using the bounding box of all the strokes in the annotation plus any additional space around the annotation where new strokes will be merged. Finally ReAnchor() is called when the anchors of the annotation need to be re-calculated. Currently, this is only used in some merging scenarios.

GetMergeBounds() returns the space around the annotation where new strokes will be combined.

The following arguments are passed to GetMergeBounds():

- annotation is the annotation to retrieve the bounds for.

GetMergeType() returns the allowed merge type. The merge type informs vsInk what to do next. There are three possible merge types: None, do not merge; Standard, merge the new stroke with the annotation; and StandAlone, generate a new Annotation instance and link it with the annotation.

The following arguments are passed to GetMergeType():

- strokeLineType is the type of new stroke being merged.

Link() connects two annotations together. This called for annotations that logically belong together (e.g. connectors and their associated note.) It is up to the implementation of IMergableStrokeTypeProcessor how linked annotations should behave.

The following arguments are passed to Link():

- annotation is the existing annotation.
Appendix C. Interfaces for Extending vsInk

- **newAnnotation** is the new annotation being linked to the existing annotation.

`Merge()` adds a new stroke to an existing annotation. While the behaviour of how this happens is up to the implementation, it typically involves segmenting and anchoring the new stroke and adding it to the list of strokes within the annotation.

The following arguments are passed to `Merge()`:

- **annotation** is the existing annotation.
- **stroke** is the new stroke to add to the annotation.
- **textView** is the *Visual Studio* code editor that the annotation is in.
- **source** is an `IRefreshable` instance that can be refreshed the merge is undone.

This method returns an `ITextUndoPrimitive` instance that allows the user to undo adding the new stroke. `StrokeMergedUndoPrimitive` is an implementation of this interface that removes the stroke from the annotation’s list of strokes.

`ReAnchor()` is called when the annotation’s anchors need to be recalculated. Currently, this method is only called when one than one annotation to an existing annotation.

The following arguments are passed to `ReAnchor()`:

- **annotation** is the annotation to re-anchor.
- **textView** is the *Visual Studio* code editor that the annotation is in.
- **anchor** is the new anchor the entire annotation needs to be anchored to.
- **anchorType** is the type of anchor to be modified.

### C.4 StrokeRendererBase

`StrokeRendererBase` provides the base functionality for rendering annotations in the code editor. Sub-classes of `StrokeRendererBase` then provide the actual implementation of how an annotation is rendered.

The following members can be defined or used in a class that implements `StrokeRendererBase`:

```csharp
public RendererMode RendererMode
```
Appendix C. Interfaces for Extending vsInk

{ get; protected set; }

public SegmentType SegmentTypeToProcess
{ get; protected set; }

public virtual IEnumerable<AnnotationStroke> RenderByLine(RenderByLineArgs args);

public virtual IEnumerable<AnnotationStroke> RenderSequential(RenderSequentialArgs args);

protected static void ChangeStrokeOpacity(Stroke strokeToRender, int opacity);

protected static DrawingAttributes GenerateAttributes(DrawingAttributes attributes, GlobalSettings settings, Annotation annotation);

protected static StrokeCollection GenerateDashedLine(Stroke stroke, int dashLength, int opacity);

protected static AnnotationStroke GenerateScaledStroke(GlobalSettings settings, Annotation annotation, AnnotationSegment segment, Stroke stroke, double newHeight, double? oldHeight = null);

protected void ClearCalculatedAnchors(IEnumerable<AnnotationSegment> segments);

When an annotation is rendered, vsInk first checks the values of RendererMode and SegmentTypeToProcess. Based on the combination of these two properties, it then calls either RenderByLine() or RenderSequential(). The two properties also determine the arguments that are passed to the method. How
these properties interact is outside the scope of this thesis; the interested reader is referred to the source code for vsInk.

RendererMode defines the base mode for running the renderer in. This will be either RendererMode.Sequential or RendererMode.ByLine. This property is set by the implementation when it is instantiated. The default value is RendererMode.Sequential.

SegmentTypeToProcess defines whether the renderer uses primary or secondary segments (see Appendix B for an explanation of the different segment types.) The allowed values for this property are SegmentType.Single, SegmentType.SegmentedRough or SegmentType.SegmentedFine. This property is set by the implementation when it is instantiated. The default value is SegmentType.Single.

RenderByLine() generates the strokes to render for a line in the annotation. This method is typically used by renderers that render the annotation in order vertically (i.e. from top to bottom).

The following arguments are passed to RenderByLine():

- `args` is the parameters for rendering the line. This includes the annotation, associated code editor, segments associated with the line and whether the line is ‘split’ from the previous line.

The method will return a set of AnnotationStroke back to vsInk. vsInk is then responsible for adding the strokes to the code editor.

RenderSequential() generates the strokes to render for a segment in the annotation. This method is typically used by renderers that render the segments of an annotation in the order they were drawn.

The following arguments are passed to RenderSequential():

- `args` is the parameters for rendering the line. This includes the annotation, associated code editor, current, next and previous segments, and whether the segment is ‘split’ from the previous segment.

The method will return a set of AnnotationStroke back to vsInk. vsInk is then responsible for adding the strokes to the code editor.

The remaining methods are helper methods for generating strokes. The following summarises the methods, the interested reader is referred to the source code for full details on these methods.
Appendix C. Interfaces for Extending vsInk

- ChangeStrokeOpacity() changes the opacity of a stroke.
- GenerateAttributes() combines the current DrawingAttributes of a stroke with the global settings to generate a new instance specific to the current rendering.
- GenerateDashedLine() generates a series of strokes to replace one single stroke so it appears that the line is dashed.
- GenerateScaledStroke() generates a new stroke that is scaled in the vertical dimension.
- ClearCalculatedAnchors() clears all the anchors that were previously calculated for one or more segments.

The StrokeRenderer attribute must be added to the class in order for vsInk to load it. The required argument for this attribute is the list of annotation classes it handles. Optional arguments include whether this is the default renderer and whether it can be activated.

C.5 AdornmentGeneratorBase

AdornmentGeneratorBase generates adornments for annotations in the code editor. An adornment is an extra visualisation that is added to an annotation. These visualisations typically provide information that assists with debugging the annotation. There are two types of adornments: for either the entire annotation or for a segment in the annotation. Implementations of this class can decide which types of adornments to provide (annotation only, segment only or both). See Table B.2 for the list of adornments already implemented in vsInk.

The following members can be defined or used in a class that implements AdornmentGeneratorBase:

```csharp
public int Order { get; set; }

public abstract IEnumerable<FrameworkElement> GenerateForAnnotation(AdornmentGeneratorArgs args);

public abstract IEnumerable<FrameworkElement> GenerateForSegment(AdornmentGeneratorArgs args);

protected static FrameworkElement GenerateLine(
    double xStart,
```
double yStart,
double xEnd,
double yEnd,
Color color,
    double thickness = 1);

_vsInk_ generates adornments in the order specified by the `Order` property. If there are two or more generators with the same order then the order is unspecified. _vsInk_ will first call `GenerateForAnnotation()` for each annotation that is visible. Then it will call `GenerateForSegment()` for each visible segment. Both methods return zero or more `FrameworkElement` instances: _vsInk_ is responsible for adding these instances to the code editor.

`Order` is an integer that specifies the order in which the generator should be run. This is set by _vsInk_ from the associated attribute.

`GenerateForAnnotation()` generates adornments for the entire annotation. This can include information like the bounds of the annotation, when it was added, its height, etc.

The following arguments are passed to `RenderByLine()`:

- `args` is the parameters for generating the adornment, including the annotation details.

`GenerateForSegment()` generates adornments for a single segment. This can include information like the bounds of the segment, the location of its anchor, the order it rendered, etc.

The following arguments are passed to `GenerateForSegment()`:

- `args` is the parameters for generating the adornment, including the segment details.

`GenerateLine()` is a helper method for generating a line element.

_vsInk_ also includes `AdornmentGeneratorAnnotationTextBase`, a sub-class of `AdornmentGeneratorBase` that generates a text adornment for an annotation. If this sub-class is used the text adornments will be automatically stacked so they do not overlap other text elements. This class defines an abstract method, `GenerateText`, that returns zero or more strings to be displayed.

The `AdornmentGenerator` attribute must be added to the class in order for _vsInk_ to load it. The required argument for this attribute is the display name
of the generator (this is the name that will be displayed in the UI). Optional attributes include the order the generator runs in and whether the generator is enabled by default.
Bibliography


BIBLIOGRAPHY


CJ Sutherland, A Luxton-Reilly, and B Plimmer. An observational study of how experienced programmers annotate program code. In J Abascal,


CJ Sutherland, A Luxton-Reilly, and B Plimmer. Location, location, location: Using spatial memory in an IDE to assist program code comprehension (work in progress). In Psychology of Programming 2016, PPiG, 2016c.


