vsInk: An Extensible Framework for Adding Digital Ink to Visual Studio

Craig Sutherland

Under the supervision of Dr. Beryl Plimmer

Dissertation submitted in partial fulfilment of the requirements of the Degree of Bachelor of Science (Honours) in Computer Science at the University of Auckland.

October 2012

Abstract

This project adds digital ink annotations into the active code editor in Visual Studio 2010. Previous attempts had failed because of limitations in the older versions of Visual Studio and Eclipse. Visual Studio 2010 provides an extended API that allows direct modification of the active code editor window. This project uses the new API to add annotations and investigates several technical challenges with combining ink and text. These include:

- Positioning strokes correctly when the document scrolled or changed
- Handling changes to the underlying code
- Handling collapsible regions
- Grouping strokes into annotations

vsInk partially solves all of these challenges. A user can add strokes to a code document. As strokes are added they are grouped together into annotations and anchored in the code. As code is changed (either by scrolling or editing) the annotations are repositioned. vsInk automatically hides annotations that are outside the editor view on in collapsed regions.

The usability study shows vsInk solves the challenges but only partially. There are two main issues: strokes are incorrectly grouped into annotations and tall annotations disappear unexpectedly. Despite these issues the subjects found vsInk easy and enjoyable to use.

vsInk opens options for future research into combining digital ink and code together and it provides a platform that can easily be extended.

Publications

Portions of this dissertation have been accepted for publication at AUIC 2013:

Sutherland, C. J. and B. Plimmer (in press). vsInk – Integrating Digital Ink with Program Code in Visual Studio. Australasian User Interface Conference - AUIC. Adelaide, Australia, Australian Computer Society, Inc.

Table of Contents

1 Introduction 1
2 Related Work 3
2.1 Early Investigations 3
2.2 Anchoring and Reflow 6
2.2.1 Grouping strokes 6
2.2.2 Recognising annotations 8
2.2.3 Anchoring annotations 8
2.2.4 Storing annotations 10
2.2.5 Reflowing annotations 11
2.3 Annotations in IDEs 12
2.4 Other Annotation Implementations 14
2.4.1 Pen-and-Paper User Interfaces 14
2.4.2 Browser-Based 14
2.4.3 Other Implementations 15
3 Requirements 17
3.1 Code Editor Integration 17
3.2 Handling Code Changes 17
3.3 Collapsible Region Support 18
3.4 Navigation 19
3.5 Extensibility 20
4 Implementation 21
4.1 Editor Integration 21
4.2 Grouping Strokes into Annotations 25
4.3 Anchoring Annotations 26
4.4 Annotation Adornments 27
4.5 Navigating Annotations 28
5 Evaluation Study 30
5.1 Methodology 30
5.2 Results 31
6 Discussion 34
7 Future Work 37
8 Conclusions 38
9 Appendix 39
9.1 Usability Test – Protocol 39
9.2 Usability Test – Participant Instructions 39
9.3 Usability Test – Questionnaire 40
10 References 42

List of Figures

Figure 1: Examples of a collapsible region in Visual Studio 2010. 19

Figure 2: vsInk in Visual Studio 2010 21

Figure 3: The Visual Studio viewport. 22

Figure 4: The layers of vsInk. 23

Figure 5: The editing surface in Visual Studio with the individual lines marked. 24

Figure 6: The process of selecting the closest line for $Line_{\#}$. 24

Figure 7: The Visual Studio editor. 24

Figure 8: The calculation from $Position_{ViewPort}$ to $Position_{Actual}$. 25

Figure 9: Example of the boundary region for an annotation. 25

Figure 10: Adornment selector window. 28

Figure 11: The hidden annotation icon and tooltip preview. 29

Figure 12: The ink navigation tool window. 29

Figure 13: Results from the questionnaires 32

List of Tables

Table 1: Recognised linker types..27
Table 2: Results of pre-test survey..30
Table 3: SUS scores for each participant...33

1 Introduction

The goal of this project is to integrate digital ink annotations with code in Visual Studio 2010. vsInk allows a user to directly annotate within the code editor. This is achieved by providing a transparent ink overlay directly in the editor window. A user can add, modify and delete digital ink anywhere on the overlay. vsInk handles scrolling and code changes by anchoring the ink to the underlying code. It also handles collapsible regions, provides navigation elements for finding annotations and allows third party extensions.

Previous research shows using a pen is a natural and easy way to annotate documents. People prefer paper because they can annotate as they read – in contrast annotating on a digital document forces the person to stop reading, add the annotation and then resume (O'Hara and Sellen 1997). The process of active reading involves annotating during read to enhance understanding (Schilit, Golovchinsky et al. 1998) – being forced to stop and annotate interrupts this process. When O'Hara and Sellen investigated paper and digital reading limitations in technology forced this mode change. Recent research by Morris, Brush and Meyers (2007) and Tashman and Edwards (Tashman and Edwards 2011) shows newer, pen-based computers allowed readers the same easy ability to annotate as paper. Pen-based computers can now allow go beyond the affordances of paper and open opportunities to improve how we read and annotate.

One commonly reviewed type of text is program code (Priest and Plimmer 2006). Program code differs from other forms of text and is often easier to write than read. Code is predominately non-linear: it is written in blocks (e.g. methods, classes, etc.) and the logical flow of a program jumps between these blocks. Program code can be printed out and annotated on paper but the non-linear nature of code makes this difficult.

A syntactical structure in modern programming languages that helps with documentation is the comment. Comments provide inline documentation within the code – including annotations – e.g. TODO comments for future work. One disadvantage of comments is they blend into the code because they are also text. Another disadvantage is they are text only – a user cannot add diagrams, drawings or other non-textual elements.

Developers use tools like Integrated Development Environments (IDEs) to understand the code. An IDE is a complex environment that contains many tools for working with code. These tools include editors with intelligent tooltips and instant syntax feedback, powerful debuggers for tracing program flow and examining state and visualizers for showing the structure of the code.

Digital ink allows a user to draw or write anywhere. Ink annotations provide a different level of visualization by letting the user quickly see digital ink it (O'Hara and Sellen 1997). But modern IDEs do not have tools for digital ink.

Previous attempts to add digital ink annotation to IDEs failed to integrate directly with the code editor (Priest and Plimmer 2006; Chen and Plimmer 2007). The main reason they failed is they could not access the internal workings of the editors (Chang, Chen et al. 2008). It was possible to access information about the editor and code but impossible to modify the actual editor. These attempts use a second window containing a copy of the code for annotations but lack the full functionality of the editor.

In this project I built a plug-in (called vsInk) for Visual Studio 2010 that allows digital ink annotations in the code editor. I first performed a literature review to find what other people have attempted with digital ink annotations – for both code and non-code documents. The plug-in uses digital ink recognition techniques to help group strokes into the annotations. It handles code changes, includes visualisations for finding annotations and offers an API for future extensions. After the plug-in was developed I carried out a usability study to find what people thought of the plug-in and to find any usability issues. The end result is a plug-in that allows for future research into how digital ink annotations and code can work together.

2 Related Work

This chapter contains an overview of related work to digital annotations. The first section looks at early research into digital annotations in general and how they might work. The second section looks at the common steps that have been identified for anchoring and reflowing digital ink annotations. The third section then looks at what previous work has been done on adding ink annotations to IDEs. The final section looks at annotations in other environments.

2.1 Early Investigations

While annotations on paper have been around for a long time (there is evidence they existed as early as 500 BC (Agosti, Bonfiglio-Dosio et al. 2007)) digital annotations are reasonably recent. This is partly because the early focus with computers was producing documents not reading them (O'Hara and Sellen 1997) and annotations are primarily part of the reading process (Schilit, Golovchinsky et al. 1998). Computer technology did not allow people to easily annotate documents, many of the affordances available with paper were lost when transferred to computers (Sellen and Harper 2003), so annotations were often ignored as “too hard” or “too inflexible”(O'Hara and Sellen 1997). One reason for this view of annotations is they needed to be text-based, like the programs and documents they were embedded in, so the computer could understand them.

Computer hardware limited early annotation systems. While research into pen computing started in the early ‘60s (Sutherland 1964) it was not commercially available until much later. With the availability of commercial pen-based computing systems researchers have been able to produce improved software systems that allow digital ink annotation and investigated how people use it.

Marshal, et al, investigated how people annotate on paper (Marshall 1997; Marshall, Price et al. 1999; Wolfe 2000; Marshall, Price et al. 2001). These investigations looked at how annotations work on paper and whether the principals can be transferred to digital documents.

In one study Marshall looked at how students annotated textbooks (Marshall 1997). Students were chosen as they do a lot of directed reading (for assignments, exams, etc.) and also a lot of annotating. Marshalled investigated both personal annotation processes and whether future readers found the annotations useful. Used textbooks in a university bookstore were analysed in the study. One key point is not all annotations are equal – in purpose or value. Marshall identified six functions for annotations:

- To clue in the reader to important material

- To serve as place marks for future work
- As an in-place way of working problems or difficult areas
- To record interpretive activity
- As a trace of the reader's attention
- As an indication of the circumstances of reading (e.g. doodles that are un-related to the content)

In another study Marshal, et al. (2001) looked at how legal students annotate in preparation for mock court. They found that people tended to annotate, review and re-annotate documents. Annotations were added during the first reading as an indication that something might be important. Students then re-read the document, scanning through to find their previous annotations, and re-annotated as necessary. Marshall, et. al. suggested annotation systems should help the user with re-reading (e.g. providing ways to help navigate between annotated sections, etc.) In addition they remarked on how annotations often varied markedly in importance, even within the same reader, so a single annotation set for all readers would fail.

One of the earliest annotation systems was a prototype called XLibris (Price, Golovchinsky et al. 1998; Schilit, Golovchinsky et al. 1998). XLibris is designed as a reading application that mimics how paper worked, including being able to annotate freely within the document. Documents are scanned (printed) into XLibris and the user views them as static pages. Annotations are linked to a location within a page. Since XLibris mimics paper it prevents changes to the documents so dynamic annotations are not needed.

While we are used to static documents on paper, computers allow us to have dynamic documents digitally. Even static documents appear different when displayed on different devices. This could be caused by different screen sizes, different aspect ratios, different available fonts, etc. If the document changes, how can annotations be moved and adjusted (reflowed) in response to these changes. XLibris was modified to handle "cosmetic" changes to documents (e.g. changing font size, page width, etc.) (Golovchinsky and Denoue 2002). The new prototype uses a set of heuristics to anchor annotations to the underlying context. These heuristics are hard-coded and difficult to extend but prove that it is possible to adjust annotations automatically.

Brush, Bargerson, Gupta & Cadiz (2001) looked at how people expect annotations to reflow. They were interested: how people reflow annotations manually; the impact of context on reflow; and what should happen to orphaned annotations. An orphaned annotation is defined as an annotation

that cannot be repositioned due to loss of context or significant changes to the document (Phelps and Wilensky 2000). Brush, et al. performed two experiments looking at reflow. In the first experiment the participants had to manually move annotations added by the researchers. This was a very difficult task as the participants misunderstood the annotations. A second test was carried out where the participants manually repositioned the annotations they had added – this was cognitively easier for the participants. Brush, et al. found the participants expected annotations to reflow "intelligently"; the participants ignore the context around the annotation. Instead they expected the system to recognise keywords and important phrases and position the annotations relatively. The participants also prefer the system to orphan an annotation (and display it as orphaned) rather than make a poor choice to the new position.

Bargeron and Moscovich looked at user expectations around automatic reflow of annotations (2003). They developed a prototype system to handle five types of annotations: underlines, highlights, marginalia, circles and margin bars. The prototype has two different types of annotation reflow: maintain the original style but move or scale as needed or clean the annotations before reflowing. For the original style, underlines and highlights extend with the line, and circles and margin bars are re-scaled. For the cleaned style the annotations are converted to a stylised form of the annotation. For example, underlines, highlights and margin bars are all converted to straight lines that precisely matched the text.

Bargeron and Moscovich evaluated what participants thought of the two styles using this prototype. Originally Bargeron and Moscovich thought the participants would prefer the original style more than the cleaned style but the results showed otherwise. Cleaned annotations raise the participants' expectations of how annotations work (for example, one participant commented the cleaning was nice but did not include all of the text that it should have!) Some issues they found were: the large variety of different annotation types, misunderstandings about the anchoring and providing feedback to the participant. The last point is particularly relevant as users were sometimes surprised at how annotations were reflowed. For this they conclude the system should provide feedback and a way to correct mistakes.

Some common themes emerge from these early studies into annotations on digital documents. One: people have different expectation of digital annotations and they are not sure of these expectations (Brush, Bargeron et al. 2001; Golovchinsky and Denoue 2002; Bargeron and Moscovich 2003). Two: people expect digital annotations to anchor and reflow "intelligently". It is not enough to do what the user does; the system should to do more and to reduce their work (Brush, Bargeron et al. 2001;

Marshall, Price et al. 2001; Bargeron and Moscovich 2003). Three: the underlying ideas are good but the technology hinders the reading and annotating process (O'Hara and Sellen 1997).

Morris, Brush & Meyers (2007) revisited the work of O'Hara & Sellen (1997) to see if reading had improved on computers. O'Hara and Sellen had concluded that people preferred paper over computers for reading for three reasons:

1. Ease of annotating
2. Flexibility in manipulation
3. Less of a context switch between note-taking and reading

Morris, et al. (2007) were interested to see whether computers had overcome these issues. They performed a similar experiment. Paper was compared to three different computing environments: a standard, dual-monitor PC; a horizontally recessed computer with pen interaction; and a three pen-based tablet system. Annotating using a pen-based computer is now as easy as using paper. When combined with features like copy-paste it is even easier. With the improved technology digital annotations are adding value compared to using paper.

This section provides background on early research into digital annotations and people's expectations. It also looks at research showing how technology now enhances the reading process. The next section looks at the issues getting annotations to reflow in an intelligent manner.

2.2 Anchoring and Reflow

The research shows a consistent set of steps to anchor annotations to a document so they reflow intelligently (Golovchinsky and Denoue 2002; Bargeron and Moscovich 2003; Ramachandran and Kashi 2003; Shilman and Wei 2004). These steps are:

i. Group strokes together into annotations
ii. Recognise (understand) the annotation
iii. Determine the anchor point for the annotation
iv. Store the annotation and associated information
v. Reflow the annotation as needed

2.2.1 Grouping strokes

When the user draws on a tablet PC the information is captured as a single stroke (Shilman and Wei 2004). While there are some single stroke annotations (e.g. an underline) an annotation usually

needs multiple strokes. To understand the annotation the strokes must be grouped together (Golovchinsky and Denoue 2002; Bargeron and Moscovich 2003; Shilman, Simard et al. 2003; Shilman and Wei 2004). There are two broad strategies for grouping – manual or automatic. For manual grouping the user selects which strokes (Bargeron and Moscovich 2003) while automatic grouping gets system to group strokes using rules or heuristics (Golovchinsky and Denoue 2002; Priest and Plimmer 2006). Automatic grouping often uses time and position data (temporal and spatial positioning). It is possible to combine manual and automatic grouping, e.g. to provide a level of automatic grouping and allow the user to "correct" the annotation if the grouping is incorrect (Chen and Plimmer 2007).

With temporal grouping strokes are a single annotation if drawn within a time period (Golovchinsky and Denoue 2002; Priest and Plimmer 2006). For example, writing a word involves a series of strokes drawn in quick succession. But the challenge is finding the right time period. Golovchinksy and Denoue (2002) analysed the times between ending one word and starting another (between-word time) and compared them to the time between strokes in a word (in-word time). They were unable to find a clear timeframe to separate in-word strokes from between-word strokes so they used 500ms. Priest and Plimmer (2006) and Chen and Plimmer (2007) increased the timeframe to 2,000ms (2s). This is because they were less interested in separating words: their primary interest was which strokes belonged to the same annotation. One point all studies have in common is temporal data is insufficient to group strokes.

With spatial grouping strokes are grouped if they are close in distance to the other strokes. Examples of this are the cross-bars on the letter 't' or the dot on the letter 'i'. Spatial grouping is needed as some people write part of the letters in a single action and then return to complete them (e.g. the cross-bars, dots, etc.) This causes a large temporal gap but only a small spatial gap (Golovchinsky and Denoue 2002). Golovchinsky and Denoue propose a set of rules based on the type of stroke, with both negative rules (e.g. if either stroke was an underline they were not merged) and positive rules (e.g. if two strokes overlapped they were merged.) A negative rule always takes precedence over a positive rule (e.g. if either stroke was an underline then they were not merged, even if they overlapped.) Priest and Plimmer (2006) also use spatial grouping but with an "annotation region". Any strokes drawn within this region are added to the existing annotation irrespective of the stroke type. Chen and Plimmer (2007) expand on the idea of a grouping region and use additional heuristics for deciding which annotation the strokes belonged to. Shilman and Viola (2004) use spatial

grouping but approach grouping as an optimization problem. They use an A* search to select which strokes should be grouped. The search space is the neighbourhood graph of closely spaced strokes.

2.2.2 Recognising annotations

The next step in the process is to recognise the type of annotation. The simplest type of recognition when the users manually identify the type (Bargeron and Moscovich 2003). While this has the highest accuracy rate it also has the highest workload for the user – it is not a very popular approach!

Several studies looked at ways of recognising the annotation type. Golovchinsky and Denoue (2002) define a set of heuristics to recognise the annotation type. They define three separate classes of annotation: annotations associated with a line of text, annotations spanning multiple lines and other annotations. Annotations are separated into these three classes based on the underlying document context. An example is when the user draws a line in the document. To recognise this requires matching the ink stroke with a line in the document. If the stroke stays within two lines in the document it is associated with the top. Some limitations of their approach include the limited number of classifications, the hard-coded heuristics and the single stroke limit.

As part of their work on adding ink to Microsoft Windows and Office, researchers at Microsoft explored different machine learning techniques recognise the annotations. First they implemented an algorithm for separating text from drawings (Shilman, Simard et al. 2003). The algorithm uses a bottom-up approach to understanding free-ink annotations. Ink is split into separate strokes and progressively combined them together. Once the strokes are grouped into annotations they are classified as either text or drawings and passed onto a separate process for recognition. Like Golovchinsky and Denoue they use a set of heuristics in the algorithm using both spatial and temporal rules but without any document context in their algorithm.

Shilman and Wei (2004) built a system for providing stable and robust anchors for reflow. They incorporated document context into their recognisers. The algorithm uses a series of recognisers that combines document context and ink strokes to classify the annotation. Each recogniser handles only one annotation type; the output of each recogniser is a likelihood rating of the annotation being of that type. A second process forms a hypothesis map and selects the best match using the map. They do not mention how the recognisers worked (e.g. heuristics, machine learning, etc.)

2.2.3 Anchoring annotations

Annotation anchoring can be divided into two broad categories:

- Co-ordinate anchoring
- Contextual anchoring

Co-ordinate anchoring uses a fixed a fixed point in a document (an XY co-ordinate). It is a simple approach but assumes the document is an image or can be treated as an image (e.g. each individual page is saved as an image.) The anchor is a single point from with the annotation: e.g. the first recorded pen contact, a corner on the bounding box, the centre of the annotation, etc. Early annotation systems (e.g. Dynomite (Wilcox, Schilit et al. 1997), XLibris (Price, Golovchinsky et al. 1998; Schilit, Golovchinsky et al. 1998) and ScreenCrayons (Olsen, Taufer et al. 2004), etc.) use this approach.

While image anchoring is very simple it cannot be used in reflow. Pages can be added or removed to a document but the page contents cannot change (Schilit, Golovchinsky et al. 1998; Golovchinsky and Denoue 2002). An extension is to generate an image from part of the document and annotate just the part (Olsen, Taufer et al. 2004). The image cannot be modified, so any changes to the document will not affect the annotation. But the annotations are now separate from the document and do not reflect any changes made.

A more complex approach is to use the underlying context from the document. This includes: lines of text, individual words, pictures and drawings and other annotations. To use context-based anchoring some understanding of the document is needed. A simple example is knowing where the individual words and lines are positioned in the document (Brush, Bargeron et al. 2001). More complex examples include dividing the document into a tree (Phelps and Wilensky 2000; Shilman and Wei 2004; Wang, Shilman et al. 2006) or a network map (Shilman, Simard et al. 2003; Shilman and Viola 2004). With a tree the document is broken into smaller blocks (e.g. document → paragraph → word → character). An item in the tree has zero or one parents and zero or more children. A network map divides the document into items that are related to each other. Tree structures are commonly used when anchoring against HTML documents as HTML itself is a tree (Ramachandran and Kashi 2003).

With context anchoring each item needs some way of uniquely identifying it (Phelps and Wilensky 2000; Shilman and Wei 2004). Phelps and Wilensky proposed three pieces of information to identify an anchor:

1. A unique identifier (if available)
2. The tree walk back to the root of the document

3. A small amount of contextual text around the anchor

These three pieces of information allow for progressive fall-back if the location cannot be identified.

Anchoring has both manual and automatic approaches. Bargeron and Moscovich (2003) use a manual approach where the user selects a context item for the anchor (typically a block of text). Brush et al. (2001) use context around an annotation to automatically generate the anchor. Shilman and Wei (2004) use the context-based recognisers to generate the anchor. This approach is extended by Wang et al (2006) and Wang and Raghupathy (2007) to include machine learning to build the recognisers. Priest and Plimmer (2006) and Chen and Plimmer (2007) use a simple set of heuristics to generate the anchor location.

2.2.4 Storing annotations

To reflow an annotation the details of the annotation (including the anchor) need to be stored somewhere. Most approaches store the annotation and anchor details in a separate document, e.g. (Ramachandran and Kashi 2003; Chatti, Sodhi et al. 2006; Priest and Plimmer 2006).

On approach is to convert the annotations into an XML format. There is a proposed standard for storing ink from W3C called InkML (Watt and Underhill 2011) which mainly deals with how to store the ink generated from a pen device. It includes elements for storing all the relevant data (e.g. pen properties, brush used, co-ordinates of the ink, etc.) and works in one of two modes: archival or streaming. Archival mode handles non real-time data that is state-free: it relies on explicitly set context. Streaming mode stores ink in sequential time order and embeds contextual information in the stream as a state change. All subsequent ink uses the new context. Neither mode stores annotation level data but it is possible to extend InkML to include it.

Other XML formats have been proposed (Ramachandran and Kashi 2003; Chatti, Sodhi et al. 2006). Some of these formats include annotation data including the anchor location. For example Ramachandran & Kashi propose a simple XML format specifically for annotations. A key point of their format is they explicitly include the anchor location as a set of references to the underlying HTML DOM elements.

Another common approach to storing ink is to use a binary format. Two examples of binary formats are Microsoft's Ink Serialised Format (ISF) (Microsoft 2007) and UniPen (Guyon, Schomaker et al. 1994). ISF is a proprietary representation of ink used in Microsoft products. It mainly stores the low level ink information in a compressed graphics format, although it does have the ability to store

metadata with each stroke. It does not group together strokes into higher level structures such as an annotation. UniPen is an open binary format that was originally proposed for storing and transmitting data for handwriting recognition. This project has since been replaced by the InkML specification and is no longer active.

2.2.5 Reflowing annotations

A review of the literature on reflowing annotations can be broadly classified into two approaches:

1. Reposition
2. Reposition and modify

Reposition is the simpler of the two approaches: the annotation is moved when the document changes. For example, if lines added or deleted above an annotation the annotation will be moved down or up (Priest and Plimmer 2006; Chen and Plimmer 2007). The main disadvantage with this approach is it does not handle changes to text underneath an annotation. For example, if the font size changes then the annotation is only re-positioned but it may not cover the original lines (e.g. if multiple lines were spanned by the annotation).

The more complex approach is to also modify the annotation. One strategy within this approach is to convert the annotation to a “cleaned” style (Bargeron and Moscovich 2003). This simplifies reflow because the annotation is be defined in precise terms (e.g. underline words two through ten.) An investigation into cleaned annotations (Bargeron and Moscovich 2003) shows users are happy with cleaned annotations and are forgiving of reflow mistakes. The downside is higher expectations of the cleaning process: they expect the system to understand their intentions (e.g. if most of the words in a sentence are underlined they expect the whole sentence to be underlined). This strategy is also limited by the annotation types the system understands. If the annotation is not recognised then the system will only reposition it. This contrasts directly with the individually of annotators (Marshall 1997; Marshall and Brush 2004).

A second strategy for reflowing and modifying annotations is to transform the original annotation while preserving the style of the original annotation (Golovchinsky and Denoue 2002; Bargeron and Moscovich 2003). Two examples of this strategy are resizing and splitting. With resizing the entire annotation is scaled to match changes in the document (e.g. if the font size has changed.) Splitting takes an annotation (e.g. an underline) and breaks it into smaller pieces, associating each new piece with the underlying context. Again this strategy depends on the system understanding the type of

annotation but it is more flexible. If the system doesn't recognise an annotation it can still apply changes like resizing when the font size changes.

One problem omitted covered by both approaches is orphaned annotations (Brush, Bargeron et al. 2001). An annotation is orphaned when its context is lost (either deleted or modified beyond recognition). Brush, et al. found people expected annotations to be orphaned when the underlying document was significantly changed. They suggest that users should be shown the orphaned annotations and a "best guess" where the annotation belongs. The user can then accept the guess or manually move the annotation.

This section covers the steps for anchoring and reflowing annotations in an intelligent way. The research has looked at annotations in general, focusing on documents produced for reading. Code is different – it is the bridge between humans and computers – and isn't read like a normal document. The next section looks at the research around annotating code.

2.3 Annotations in IDEs

There are a few projects that added ink annotations within IDEs. The first attempt is Rich Code Annotation Tool (RCA) (Priest and Plimmer 2006). RCA adds ink annotations to Visual Studio 2005. Three approaches were explored for adding ink:

1. A transparent overlay within the editor window
2. An associated "design" window
3. A separate tool window for the ink

The first two approaches could not be implemented due to a limitation in the IDE. Visual Studio 2005 prevents third-party developers from modifying the code editor. To work around this limitation the user had to open a new window that contained a read-only copy of the code. The user annotated the code in the new window. There are two draw-backs to this approach: the user cannot modify the code in the annotation window and they cannot use the rich functionality provided by the default editor.

RCA uses the concept of a linking stroke for anchoring annotations. RCA provides two modes: a linking mode (starts new annotations), and an inking mode (modifies existing annotations). The first stroke becomes the anchor for the entire annotation. RCA associates the anchor with a code line based on the stroke type. Line strokes use the line closest to the starting point is as the anchor. Circle strokes use the line closest to the mid-point of the stroke. After the linking stroke is added RCA

switches to inking mode. The user can also switch to Inking mode by selecting on an existing annotation. RCA stays in inking mode until the user does not draw for two seconds then it returns to linking mode.

CodeAnnotator is the second attempt report, it uses Eclipse for the IDE (Chen and Plimmer 2007). Eclipse is an open source IDE so the researchers thought they would have more success as they could modify the IDE if needed. However CodeAnnotator has the same problems with IDE extensibility and also uses a separate window for the ink annotations. The researchers explored modifying the IDE code but decided against it as the code is very complex. CodeAnnotator uses the linker concept with extended rules that use spatial positioning for grouping strokes. If a stroke is within the bounding box of an existing annotation then it is merged with the existing annotation.

These attempts identify some common limitations in the extensibility models for Visual Studio 2005 and Eclipse (Chang, Chen et al. 2008). The first issue is the lack of extensibility points for the code editors. It was possible to retrieve information from the editor but not to alter the user interface. The second problem is a number of core services (e.g. language services) are un-reachable. This meant they would have to be re-written to use them. Finally there are issues with the scroll bars incorrectly firing as they sometime fail to fire when the user scrolls, which makes it hard to synchronise code and annotation layers.

There are two further tools that add annotations to IDEs – CodeGraffiti (Lichtschlag and Borchers 2010) and Collage (Bradley 2012). CodeGraffiti extends the XCode IDE (the development IDE for Apple devices). It links annotations to one or more anchors within any text (not just code) with persistence. It also allows annotations on other document types (e.g. screen elements, widgets, etc.) but these annotations are not anchored or persisted. It is not mentioned whether a user can annotate directly on the code. Collage is an open source project that extends Eclipse to add drawing tools to the editor. Unlike both RCA and CodeAnnotator it works in the actual code editor window, allowing a seamless integration of both code and drawings. It works in a similar way to popular graphical editor tools like PhotoShop and Gimp by adding one (or more) drawing layers. While not specifically focused on annotations it provides similar functionality with the free-hand drawing tool. Drawings are anchored to the text lines so they are reflowed when the underlying document changes (although this is repositioning only, it does not attempt any modifications).

This section reviews the projects that around integrating ink annotations in IDEs. There are few studies in this specific area but more studies that look at ink in other areas. The next section looks at some examples.

2.4 Other Annotation Implementations

This section considers three broad categories of non-IDE ink applications: pen-and-paper user interfaces, browser interfaces and other interfaces. Pen-and-paper user interfaces use real paper with computer systems and attempt to combine the affordances of paper together with the power of computers. Browser interfaces allow users to add digital ink when browsing the internet. Other interfaces look at various other implementations in a variety of contexts.

2.4.1 Pen-and-Paper User Interfaces

The pen-and-paper user interface uses a special pen with paper and converts the strokes into digital strokes (Liao, Guimbretière et al. 2005; Steimle, Brdiczka et al. 2008; Brandl, Richter et al. 2010). This approach combines the affordances of using real paper with the advantages of digital ink. The pen uses a special pattern on the paper to detect the location where the user is annotating and either stores the data or wirelessly transmits to a local computer. Since the document is printed on paper it is effectively a static document at the time of annotating which reduces some of the complexities with digital annotations. The main challenges with pen-and-papers user interfaces are merging annotations with the digital documents and providing feedback to the user.

PapierCraft (Liao, Guimbretière et al. 2005) is a system that treats paper documents as proxies for a digital document. The user draws anywhere on the page and the annotations are merged back to the digital document. The system provides gestures to allow editing actions (e.g. cut-and-paste.) When the ink is merged the gesture commands are replayed using a digital interface to show the user see the effect of the commands: the user can accept or change the commands as needed.

CoScribe (Steimle, Brdiczka et al. 2008) is a collaborative annotation system for lectures. Students print out the lecture slides and annotate them during a lecture. After the lecture they upload their annotations, which are automatically synchronised to all students. Students use a computer to view the merged annotations. While CoScribe handles multiple sets of annotations it assumes that the underlying document is static and does not change.

2.4.2 Browser-Based

Most browser applications fall into two categories: text-based annotation systems or ink-based annotation systems. One limitation of browsers is they do not natively support pen input: pen input

is converted to mouse input. To get around this additional plug-ins are required for the browser – Adobe Flash (Chatti, Sodhi et al. 2006) and Microsoft Silverlight (Plimmer, Chang et al. 2010) have both been investigated. An alternate approach is to ignore the additional information provided by pen systems and use the mouse data instead (Ramachandran and Kashi 2003).

A second limitation of browser based annotating is they cannot modify the document being annotated. Therefore annotations cannot be stored within the document: they need to be stored in a separate location. *u*-annotate (Chatti, Sodhi et al. 2006) stores the annotation in local storage in an XML format, which can be uploaded to an annotation server. iAnnotate (Plimmer, Chang et al. 2010) does not mention how or where the annotations are stored.

Another browser-based implementation is NB (Zyto, Karger et al. 2012). NB allows students to collaboratively annotate a PDF document. The documents are converted the document to images (one per page.) The user draws a region on the image and types in an annotation. The pages are static images with absolute positioning within each page. There is no attempt to reflow at all. NB functions as a client-server application: data is sent from the browser and stored on the server.

2.4.3 Other Implementations

In addition to the systems mentioned above there have been a few other examples of ink-based annotation systems. Some examples are LiquidText, ScreenCrayons, Classroom Presenter, and PenMarked. This section provides a quick overview of each of these systems and describes how they handle grouping and anchoring.

LiquidText is a tool that supports active reading (Tashman and Edwards 2011; Tashman and Edwards 2011). LiquidText explores some ways that an electronic reading system can overcome the limitations of paper-based reading. While it is mainly focused on helping the reading process it also provides some functionality to help with annotations (this is normally considered part of "active reading"). To achieve this LiquidText provides the ability to "extract" content and annotations and to associate annotations with multiple sections. While not mentioned directly it appears that LiquidText only handles static documents, so it does not need to handle any document modifications. Annotations are associated with one or more blocks of text. One of the more interested features in LiquidText for annotations is they are treated like any other excerpt, allowing for full control over how the annotations can be positioned, arranged and grouped. However LiquidText only use text-based annotations. The user needs to manually type in the annotation (using an on-screen keyboard). Therefore these are not true digital ink annotations.

ScreenCrayons is designed as a generic screen capture and annotation system (Olsen, Taufer et al. 2004). The main design goal was a system that could be used to annotate anything. This is achieved by allowing the user to do a screen capture of any application that is running. After the screen capture the user can add any annotations they like. The actual capture uses free-form ink. Since the actual artefact being annotated is a static image anchoring is handled using co-ordinate anchoring and there is no attempt to handle any updates to the underlying documents. ScreenCrayons uses text recognition to help with recognition of the various strokes, but this is mainly to assist with navigation through the annotations. There is no mention of any grouping in ScreenCrayons.

Classroom Presenter is a replacement for PowerPoint that enhances annotations during lectures (Anderson, Anderson et al. 2004; Anderson, McDowell et al. 2005; Anderson, Davis et al. 2007). It allows a lecturer to annotate slides using a tablet PC and the slides and annotations are displayed in real-time. Later versions also allowed students with tablet PCs to annotate the slides as well and the lecturer can see the results and (if desired) display them to the rest of the class. Since Classroom Presenter was designed for real-time annotations it does not include any grouping or anchoring.

3 Requirements

The previous chapter provided an overview of digital ink, annotations and anchoring from the literature. This chapter draws on the related work to specify the requirements for this project.

3.1 Code Editor Integration

The user should be able to annotate code within the active editor and it should be as simple as changing to annotation mode and then drawing in the editor. This is in contrast to previous attempts that required annotations to be in a separate window with a read-only copy of the code (Chang, Chen et al. 2008). The users should still be able to use all of the functionality that is provided by the editor and also add annotations.

The plug-in should integrate directly within the IDE to reduce confusion. All toolbars and windows should have the same look-and-feel as the Visual Studio 2010 IDE. This is to help the users learn the plug-in as different UI elements may confuse users who are familiar with Visual Studio 2010.

The strokes need to be grouped together into annotations. This is needed for both reflowing annotations and integrating the annotations with navigation elements. The grouping should be in a natural appearing way: vsInk should group strokes in the same way the person adding the strokes would group them.

For grouping the rules defined for RCA and CodeAnnotator (Priest and Plimmer 2006; Chen and Plimmer 2007) should be the starting point. (Temporal and spatial checks should be used to check whether new strokes are part of an existing annotation.) These rules will also be expanded to handle additional scenarios, e.g. for choosing between annotations selected by other rules.

There are four actions needed for annotations: adding, modifying, moving and deleting. A new annotation is added (started) with a stroke that is not part of any existing annotation. A modification occurs when a stroke is grouped with an existing annotation or if an existing stroke is erased. A move occurs when a stroke within an annotation is selected and moved – this should move the entire annotation and generate a new anchor point. A deletion occurs when all the strokes within an annotation are erased – this should remove the annotation and all associated data.

3.2 Handling Code Changes

vsInk should handle any changes to the underlying code in the editor. There are two types of relevant changes: changing the view and editing the code. A view change is when the displayed code is changed, e.g. the user scrolls through the code, and when the editor is opened for a new

document. A code modification is when the code is edited: either with the editor or by an external source. Due to the nature of the anchoring (see below) only two types of changes should be handled: line insertions and line deletions.

This requires an anchor point for every annotation. The anchor point is needed for locating the annotation relative to the code. vsInk should use linking stroke concept from RCA (Priest and Plimmer 2006). The first stroke in an annotation is the linking stroke and all other strokes are positioned relative to this stroke.

The position of the linking stroke should be calculated using the anchor single point for the stroke. The actual anchor point depends on the type of linking stroke (e.g. for a line the left-most point is used, for a circle or brace the middle of the left border is used, etc.) RCA defined two types of linking strokes (a line and a circle) with some heuristic rules to determine the type. The hard-coded heuristic rules makes it hard to add new linking types so vsInk should use Rata.Gesture (Chang, Blagojevic et al. 2012) for classifying the strokes. Rata.Gesture allows us to build classifiers using machine learning allowing for a greater range of linking types and adding more types as needed.

vsInk should handle code changes by repositioning annotations after every code edit. If lines are inserted or deleted above the anchor point the annotations should be moved down or up by the same amount. There is one exception to this – if the lines containing the anchor point are deleted then the entire annotation should be deleted.

3.3 Collapsible Region Support

Most modern IDE support collapsible blocks of code (these are called regions in Visual Studio). A collapsible block can be used to hide parts of the code to help the developer focus on the relevant code (see Figure 1). Visual Studio automatically generates regions for various code structures (e.g. methods, properties, classes, doc-comments, etc.) and allows the user to define their own regions using the #region/#endregion syntax.

```
444  Delete()
456
457  #region Clone()
458  public ViewModel Clone()
459  {
460      var model = new ViewModel { outputPa
461      this.CopyTo(model);
462      return model;
463  }
464  #endregion
465
466  CopyTo()
```

```
444  Delete()
456
457  Clone()
471
472  CopyTo()
```

Figure 1: Examples of a collapsible region in Visual Studio 2010.

Top view: an expanded region; bottom view: the same region is collapsed.

vsInk should handle collapsible regions in two ways. Collapsing or expanded should be handled as a code modification for any annotations outside of the region (annotations should stay fixed relative to their anchor points). Any annotations with an anchor point inside the region should be hidden or restored. Folding annotations (hiding only part of an annotation) will not be implemented for this project.

3.4 Navigation

Users need to both add annotations and later find them. This can be difficult as the file size or number of regions increase and navigating to annotations in collapsed regions can be challenging as Visual Studio allows for regions within regions (and these can in turn be inside other regions, etc.). To find an annotation within a region the user would need to expand all the relevant regions down to the region that contains the annotation's anchor point. To handle this vsInk should add an indicator to the editor showing the user that there is an annotation in a collapsed region. This should work in a similar way to how Visual Studio shows collapsed regions. A small icon should be added on the same line. When the user clicks on this icon all the relevant regions should be expanded so the annotation is visible.

To handle large file sizes an annotation navigation window should be added. This window should display a preview of all the annotations in a code file. Clicking on an annotation preview should scroll the editor to the relevant line in the code and expand all necessary regions. To help with navigation the previews should be sortable by either position within the file or the time when the annotation was added.

3.5 Extensibility

vsInk should allow for extensions without needing to rewrite the code. While there are a large number of possibilities for how people might want to extend vsInk the following extension points should be added:

1. New linking strokes types can be defined. This should be as simple as adding the new model to vsInk and defining where the anchor point is relative to the stroke.
2. Adding new adornments to the ink canvas. These should allow new information to be displayed for annotations. In addition these adornments should be configurable by the end user (e.g. whether they are visible, how visible they are, etc.)
3. There should be an API exposed to allow for further extensibility options. This should utilize the same Managed Extensibility Framework used by the rest of Visual Studio.

4 Implementation

vsInk is implemented using C#, WPF and the .Net 4 framework. It uses the Visual Studio 2010 SDK for integration with Visual Studio 2010 and consists of a single package that is installed into the IDE. It is designed to be used on a Tablet PC but can also be used with a mouse on any Windows PC. Figure 2 shows the main elements in the user interface.

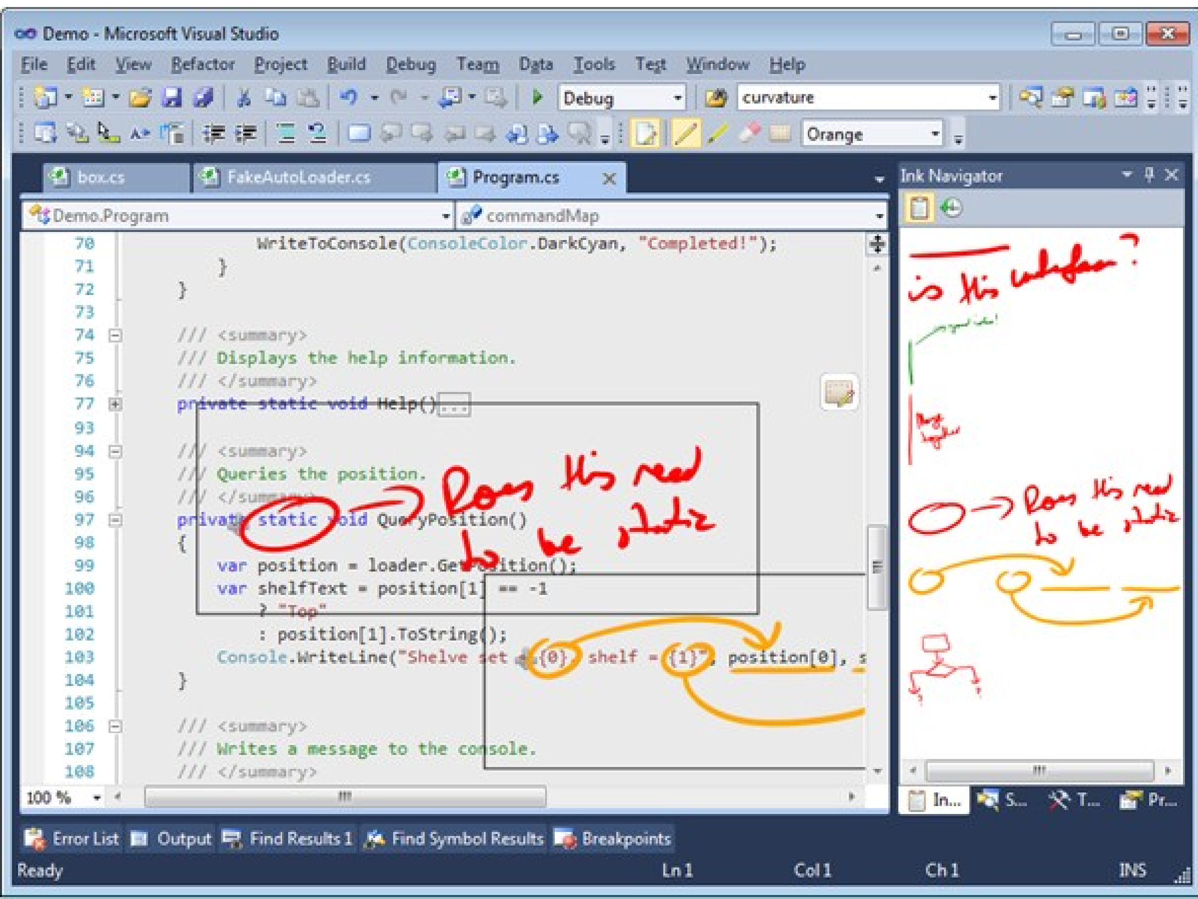

Figure 2: vsInk in Visual Studio 2010

This section describes the five major features of vsInk: editor integration, grouping annotations, anchoring annotations, annotation adornments and navigation.

4.1 Editor Integration

The Visual Studio 2010 editor includes adornment layers. These layers allow plug-ins to add visual elements directly within the code editor. A plug-in defines one or more layers and their relative position (z-order) and Visual Studio renders them in the actual editor.

Visual Studio defines three types of adornment layers: text-relative, view-relative and owner-controlled (Microsoft). Each layer positions the adornments in slightly different ways. A text-relative layer expects each adornment to be associated with a text element: Visual Studio automatically

positions the adornments relative to the text element. A viewport relative layer generates a layer that is positioned relative to the viewport of the editor. This viewport includes all the code currently displayed plus it also includes some lines above or below the viewable space and is as wide as the widest line (see Figure 3). The adornment layer is positioned relative to the viewport and is updated whenever the viewport changes. An owner-controlled layer is positioned relative to the actual editor window and is not changed by Visual Studio.

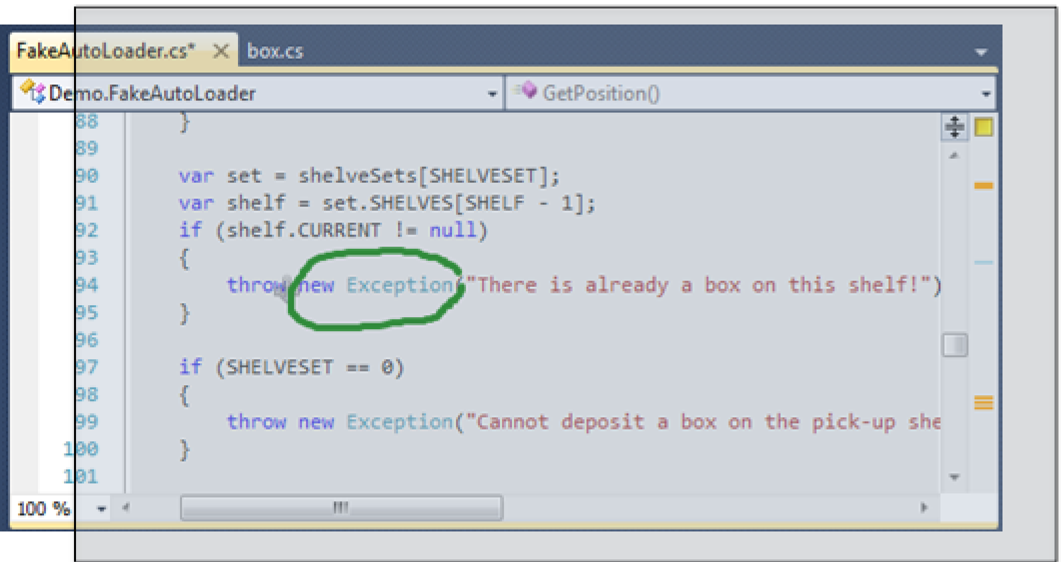

Figure 3: The Visual Studio viewport.

Of the three types viewport-relative was initially the most promising. The text-relative layer required an association with an element of text but vsInk needs a single visual element to cover the entire viewport. Since viewport-relative offered this I thought it would a good match. However there were a number of scenarios where the scrolling broke (e.g. when moving to the top or bottom of a long file) which caused by the way Visual Studio regenerates the viewport on the fly. Various attempts to fix these issues failed so the viewport-relative approach was abandoned.

Finally I tried the owner-controlled adornment layer. This layer type does not have any automatic positioning: it expects the plug-in to position the items as needed. This involves more work but it allows full control over where the elements are displayed. Initially a single ink canvas was added to the editor to contain both ink strokes and other visual elements. The ink canvas was positioned to cover the entire viewpoint – from the top-left corner to the bottom-right (since the viewport in Visual Studio is more than just the viewable screen area).

While the single ink canvas approach works it has some unintended side-effects. The user can select and modify the visual elements as well as the strokes! Therefore I replaced the ink canvas with a custom three layer control. The bottom layer (adornment layer) is a standard WPF canvas that contained the majority of the non-ink visual elements. These elements can still be seen but they cannot be selected. This is covered by an ink canvas. This canvas allows the user to add, modify and delete ink strokes. Since the visual elements are separate the user cannot select or modify them. Since the ink canvas is on top of the adornment layer the user cannot interact with the adornments either. Since some navigation elements need interaction (e.g. clicking, tool-tips, etc.) a third layer is added on top of the ink canvas (the navigation layer). This layer contains the visual elements related to navigation (see Figure 4).

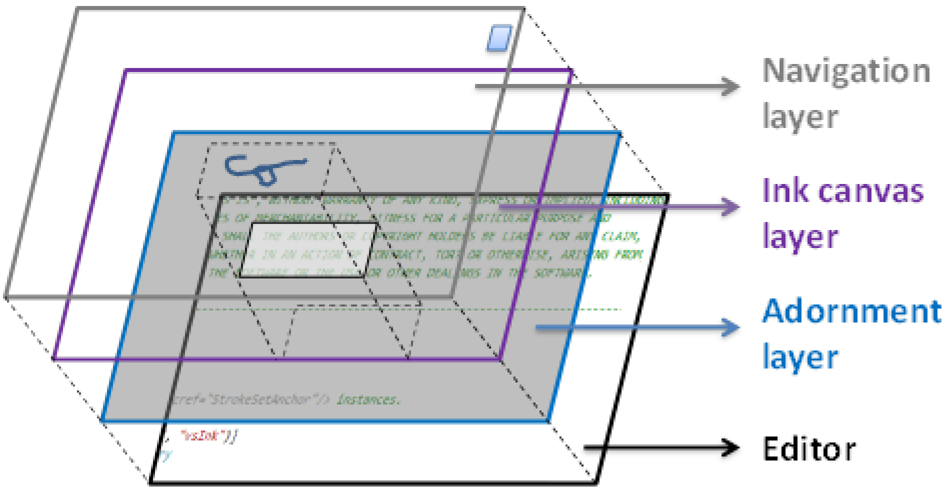

Figure 4: The layers of vsInk.

Since the adornment layer is owner-controlled vsInk positions all the annotations itself. vsInk listens for any viewport changed events (Visual Studio fires these whenever the user scrolls through the document). Every annotation is updated to a new position based on its annotation anchor. The annotation anchor is based on two pieces of information: the line number ($Line_{\#}$) and offset ($Offset_{Line}$). The editor in Visual Studio is broken up into a number of lines (see Figure 5). When a new annotation is started the closest line to the linker anchor point is selected (see Figure 6) as $Line_{\#}$. Internally $Line_{\#}$ is recorded as a Visual Studio tracking point; this enables vsInk to use Visual Studio's automatic line tracking to handle any changes to the code.

```
91    var shelf = set.SHELVES[SHELF - 1];
92    if (shelf.CURRENT != null)
93    {
94  LineHeight  ew Exception("There is alrea
95    }
96
97    if (SHELVESET == 0)
```

Figure 5: The editing surface in Visual Studio with the individual lines marked.

```
91    var shelf = set.SHELVES[SHELF - 1];
92    if (shelf.CURRENT != null)
93    {
94      throw new Exception("There is alrea
95    }
96
Line#  if (SHELVESET == 0)
```

Figure 6: The process of selecting the closest line for $Line_{\#}$.

As the user scrolls through a document Visual Studio fires viewport changed events (these events are also fired when Visual Studio regenerates the viewport). vsInk listens for these events and updates the annotation positions. The update handles setting the visibility and position of each annotation. There are two visibility checks: whether $Line_{\#}$ element is visible and whether $Line_{\#}$ is inside a collapsed region. If either checks passes the annotation is hidden; otherwise it is visible. If an annotation is visible a translation is applied to move it to the correct position.

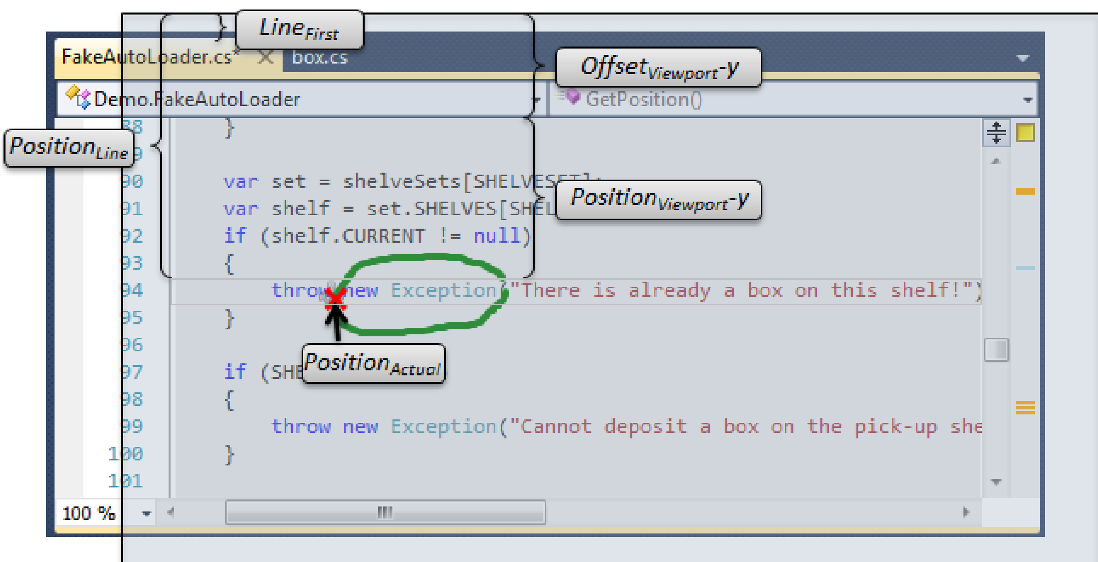

Figure 7: The Visual Studio editor.

The grey box shows the viewport region used by Visual Studio.

Positioning of an annotation requires several steps. First the line number of the first line in the viewport ($Line_{First}$) is subtracted from $Line_{\#}$ to give the offset line number ($Line_{Offset}$). $Line_{Offset}$ is multiplied by the line height ($Line_{Height}$) to get the line position ($Position_{Line}$) in the adornment layer. The viewport offset ($Offset_{Viewport}$) is subtracted from $Position_{Line}$ to get the viewport-relative position ($Position_{ViewPort}$) (see Figure 7). Finally $Offset_{Line}$ is added to $Position_{ViewPort}$ to get the actual position

($Position_{Actual}$) (see Figure 8). $Position_{Actual}$ is used to translate the annotation into its correct position on the canvas.

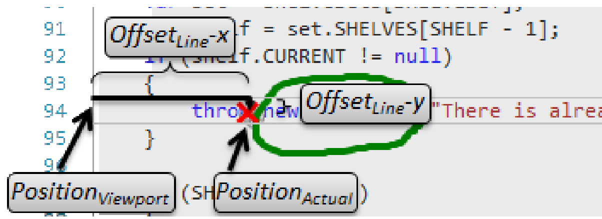

Figure 8: The calculation from $Position_{ViewPort}$ to $Position_{Actual}$.

Collapsible region support is handled by listening to the relevant events in the editor. The region manager in Visual Studio fires three events for region changes (*RegionsChanged*, *RegionsCollapsed* and *RegionsExpanded*). When vsInk receives any of these events it updates the ink canvas (the same as when the viewport changes).

Code changes are handled in a similar way: the only difference is the event that starts the update. Since vsInk uses tracking points the actual line is calculated by Visual Studio. All vsInk does is listen to the code change events and performs the same canvas update. There is one additional check for delete events – the entire annotation is deleted when the anchor point is deleted.

The annotation ink and associated elements are serialised to a binary format and saved automatically. The strokes are stored in the Ink Serialised Format. When a new document window is opened vsInk loads the annotations if an existing ink file exists.

4.2 Grouping Strokes into Annotations

Grouping annotations is a problematic process: one rule was used for grouping but this had a high error rate. The initial process used a bounding region around the annotation: this was 30 pixels around the bounding box of the annotation (see Figure 9). Each new stroke was against existing annotations and if it intersected any existing boundary region it was added to the annotation. If the stroke intersected multiple annotations it was added to the first annotation found. If the stroke did not intersect any existing annotations it started a new annotation.

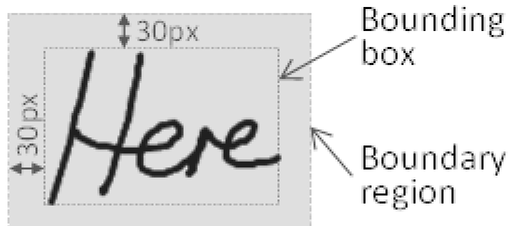

Figure 9: Example of the boundary region for an annotation.

Usability testing (see below) showed that this was inaccurate. The boundary region was too wide – ink stroke was too thick and the code lines too close. Other problems included: the strokes not added to a word annotation and the wrong annotation being selected when the stroke intersected multiple annotations.

I made three changes to address these issues: the boundary was decreased to 20 pixels; a timing check was added to handle writing a sentence; and an improved multiple annotation check added. I increased the boundary to handle the thickness of the brush. The timing checked added the stroke to an annotation if it is added within 0.5 seconds of the last stroke. The literature reports two numbers for temporal grouping – 0.5 seconds (Golovchinsky and Denoue 2002) and 2 seconds (Priest and Plimmer 2006). Both were trialled and 0.5 seconds was found to be more accurate for grouping. The new multiple annotation check chooses the annotation with the closest middle point to the starting point of the stroke. Euclidean distances are used to calculate the closet middle point.

4.3 Anchoring Annotations

vsInk uses linking strokes for the anchor in a similar way to RCA and CodeAnnotator (Priest and Plimmer 2006; Chen and Plimmer 2007). In vsInk the linking stroke is the first stroke of a new annotation. The anchor point is using the type of the first stroke. RCA and CodeAnnotator used simple heuristics for determining the linking stroke type (a line or circle) but this is too limiting as new types are needed. I used Rata.Gesture (Chang, Blagojevic et al. 2012) to recognise the stroke type. Rata.Gesture is a tool developed at the University of Auckland for generating ink recognisers. Rata works by extracting 115 features for each stroke and using machine learning to train a classification model. vsInk uses a model generated by Rata.Gesture for classifying the linking stroke.

I carried out an informal user survey to collect the most common types of linking strokes. From this list I selected six types (see Table 1). Ten users provided ten examples of each stroke: giving a total of 600 strokes to train the model.

When a new annotation is started the recogniser is used to classify the type of linking stroke. Each type has a specific anchor location used to calculate $Line_{\#}$ for anchoring.

Table 1: Recognised linker types.

The red cross indicates the location of the anchor.

Linker Type	Example	Linker Type	Example
Line – horizontal		Line – vertical	
Line – diagonal		Circle	
Brace		Arrow	

4.4 Annotation Adornments

vsInk allows each annotation to have associated adornments. These adornments are different from the Visual Studio adornments in two ways: they are associated with an annotation and their visibility is controlled by vsInk. vsInk has two default adornments in vsInk: the boundary region indicator and the anchor indicator. Third parties can add new custom adornments. An example of a custom adornment is provided in the project that displays the name of the user who added the annotation.

A user can enable or disable adornments in vsInk by using the adornment selector window (see Figure 10). The selector lists all the available adornments that are loaded. The user enables or disables an adornment using the checkbox next to each adornment. Changing the selected adornments automatically updates the adornments in the editor.

The selector window also allows setting the transparency of the adornments. This is a global setting for all adornments: it controls the transparency of the adornment sub-layer.

Adding an annotation calls a factory class to generate the adornments: for loaded annotations (e.g. when a document is opened) and user-added annotations. Each adornment is added to adornment sub-layer. Custom adornments are added to vsInk by adding a new factory class.

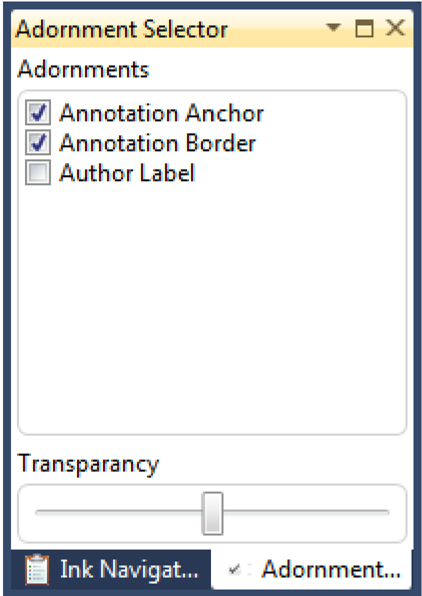

Figure 10: Adornment selector window.

Adornments are positioned using a similar process to ink strokes. If an annotation is hidden during a canvas update all the associated adornments are hidden as well. If the annotation is visible then each adornment for the annotation is called to update its location. Adornments typically update their position using the details from the annotation (e.g. the bounding box or similar).

4.5 Navigating Annotations

There are two types of navigation: collapsed region support and navigation outline. Collapsed region support adds an icon to the navigation sub-layer there are annotations in collapsed regions. The icon is added or removed during the canvas update process. This ensures the icon is always up-to-date irrespective of the changes to the editor.

Clicking an icon automatically expands the collapsed regions containing the annotation and scrolls the editor so the annotation is in view. The annotation is "flashed" to indicate where it is. "Flashing" is implemented by the different adornments. The default implementation changes the border size on the boundary region indicator. When the user hovers over the icon a thumbnail is displayed of the entire annotation (see Figure 11).

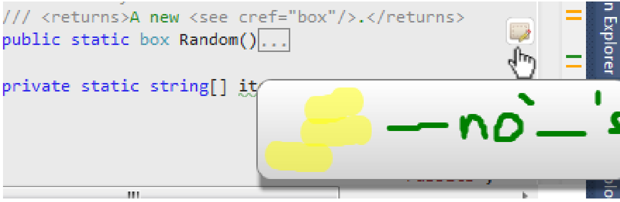

Figure 11: The hidden annotation icon and tooltip preview.

The navigation outline is a separate tool window in Visual Studio. This gives the user full control over where the window is positioned. The window contains a scrollable list of a thumbnail of each annotation in the current document (Figure 12). Each annotation is scaled between 25% and 100% of the original size to fit as much of the annotation as possible in the thumbnail.

Figure 12: The ink navigation tool window.

The user can switch between position and timeline views of the annotations by changing the sort order (the two buttons on the top of the window). The position view uses the line number for sorting. The timeline view uses the time the annotation was first added.

The navigation view can be used to navigate to an annotation by clicking on it. This scrolls the window, expands any collapsed regions and “flashes” the annotation.

5 Evaluation Study

I performed a task-based usability study to evaluate the effectiveness of vsInk. Subjects performed two code review tasks and annotated any issues found. Usability information was collected via researcher observation, questionnaires and informal interviews. This section describes the methodology of the study and then the results.

5.1 Methodology

There were eight participants in the study (6 male, 2 female): four were computer science graduate students, three full-time developers and one computer science lecturer. All had experience with Visual Studio: most participants say they use it frequently (see Table 2). Participants were evenly split between those who had used pen-based computing before and those who hadn't. All but one of the participants had prior experience reviewing program code.

Table 2: Results of pre-test survey

Question	Always 1	2	3	4	Never 5	Median
I use Visual Studio 2010 for development	4	2	2	0	0	1.5
I use Visual Studio 2010 for reviewing code	1	1	4	1	1	3
I use digital annotation tools for reviewing documents	0	1	1	3	3	4
I have used pen input on a computer	1	2	1	3	1	3.5
I review and write annotations on my own code	2	4	1	0	1	2
I review and write annotations on other people's code	0	2	2	2	2	3.5

The usability test had two rounds of testing. After the first round of testing the major flaws were fixed and a second round performed.

Each study started with a pre-test questionnaire to gauge the participant's previous experience with the tools and tasks. The researcher then showed vsInk to the participant and explained how it worked. The participant was allowed three minutes to familiarize themselves. In the first task the participant was given a set of simple C# code guidelines (eight in total) and a small application consisting of four C# code files: they were asked to review the code and annotate where the code did not conform to the guidelines. Each participant was given eight minutes for this task (they could finish earlier if desired). After the task an automatic process updated the code and the participant reviewed the code. During the review they were asked to find the previous annotations and update

them stating whether the issue was fixed or not (not all issues were fixed). The researcher observed the participant and noted down any usability issues.

The participants filled in a questionnaire after each task. At the end of the tests the researcher and participant went through all annotations and identified whether the annotation was correctly re-positioned.

After this the researcher performed an informal, semi-structured interview. The main purpose of the interview was to find out what each participant liked and disliked about vsInk.

5.2 Results

After the first four subjects the results were reviewed and the issues identified: some were fixed before the second round of testing. The main issue was strokes being incorrectly added to annotations. The opposite (strokes not being added to annotations correctly) occurred rarely. Therefore three changes were made to the grouping process (see 4.2 above).

There were other refinements in addition to the grouping changes. Some participants complained that the lines were too small and the ink too fat: the ink thickness was reduced and the line size increased slightly. Another complaint was the adornments obscured the code: all adornments were made semi-transparent and non-necessary adornments removed (e.g. the name of the annotator). Participants mentioned they did not recognise the annotations in the ink navigator: the amount of distortion was limited to between 20% and 100% of the original. Participant did not know which annotation they had selected in the navigator: a "flash" was added to clarify which annotation was selected.

The issues not fixed were: tall annotations disappearing when anchor point out of viewport, cut/copy and paste not including annotations, and annotations not included in undo history.

After the modifications the second set of participants tested vsInk. Most of the modifications had the desired effect and vsInk was easier to use. But there are still issues with stroke grouping. Using time in grouping sometimes caused strokes to be added incorrectly, especially when moving quickly down the code. The boundary region still causes problems when starting a new annotation if the lines of interest are close together.

Together the researcher and participants identified a total of 194 annotations. Each participant added a mean of 28 annotations – a t-test found no evidence for any different between the two

rounds of testing (p-value > 0.05). Of the 194 annotations 57 (29%) were incorrectly repositioned after the code update. While this dropped from 36% in the first round to 26% in the second round a t-test found no evidence of this being statistically significant (p-value > 0.05). The majority of the incorrectly positioned annotations (52 out of 57) were as a result of grouping errors.

The questionnaire asked the participants to rate vsInk on a number of features (see Figure 13). The questionnaire consisted of a number of statements using a 5-point Likert scale. The subjects were asked whether they agreed with the statement (score = 1) or disagreed (score = 5).

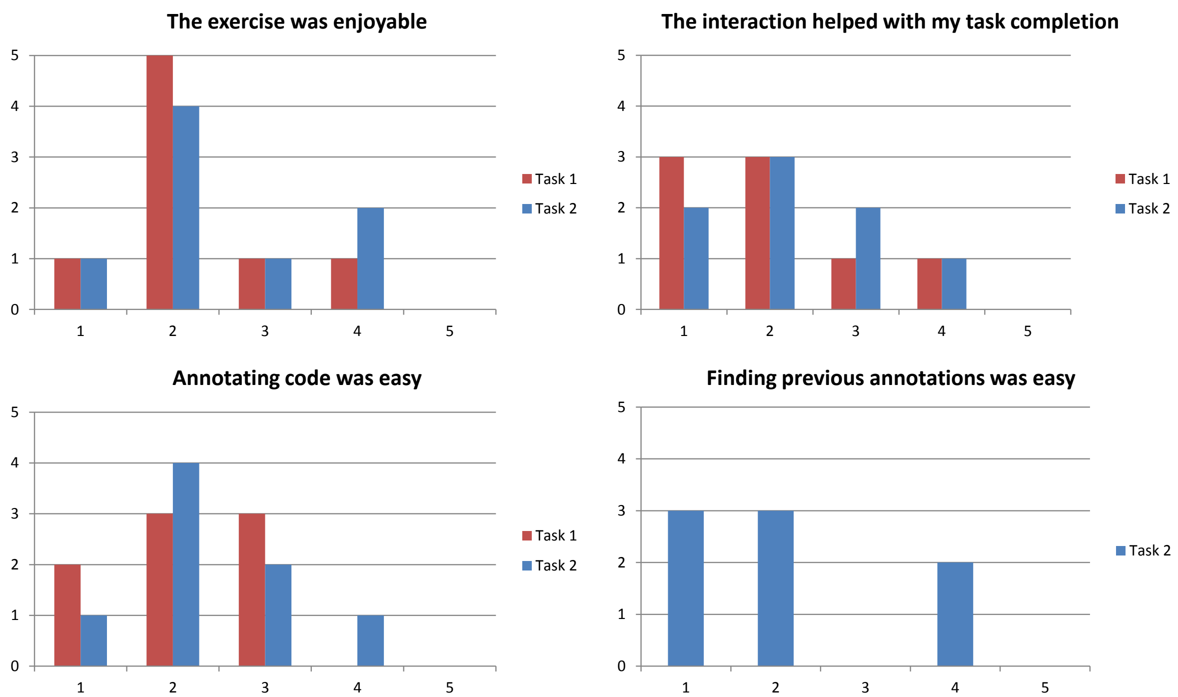

Figure 13: Results from the questionnaires

The majority of the participants agreed that the exercise was enjoyable, using vsInk helped complete the task and annotating code was easy. There appears to be a slight drop in agreement for all three statements in the second task but a Mann-Whitney U-Tests found no evidence of any difference any of the statements (p-value > 0.05). Finally the majority of the participants agreed that it was easy to find annotations in the second task.

At the end of the session the subjects filled in a System Usability Scale (SUS) survey (Brooke 1996) - see Table 3. The mean SUS score was 74.4 with a standard deviation of 13.0. Based on the results from Bangor, Kortum & Miller (2008) this rates the usability of vsInk as about average compared to other GUI-based applications.

Table 3: SUS scores for each participant.

Participant	1	2	3	4	5	6	7	8
Score	47.5	75	77.5	87.5	77.5	65	77.5	87.5

During the informal interviews most subjects stated they liked how vsInk provided the freedom to do any annotations they wanted. There was also general agreement that ink annotations stood out better than inline text comments and were much faster to find. However some subjects found that the annotations obstructed the underlying code, making it harder to read. While most of the participants understood why vsInk grouped the strokes together they thought the grouping routine was too inaccurate. In addition to improving the grouping the some suggested improvements were being able to selectively hide annotations (maybe by colour), having some form of zoom for the code under the pen and displaying the code in the navigator window.

6 Discussion

With vsInk I integrated digital ink directly into the Visual Studio code editor. Ink strokes are grouped together into annotations and positioned correctly the code is scrolled or edited. At the same time all the existing functionality of the editor is still available to the user. While this is now possible there were a number of challenges that needed to be overcome and some still outstanding!

The initial technical challenge was how to maintain the correct positions of the annotations relative to the code. While the viewport-relative adornment layer initially looked promising the actual implementation caused problems. Visual Studio appears to have two different update modes – one mode for when the user scrolls within an existing viewport and another for when the viewport needs to be regenerated. Unfortunately there is no way to tell the difference between these modes and the adornment layer behaves in different ways depending on the mode. For vsInk the main problem is sometimes Visual Studio would report that the top of the viewport is within the viewable screen space – giving the effect of having a region above the viewport that is seen by the user. Other times it would give incorrect views for where the top of the viewport was (e.g. saying the top of the viewport was line 1 when it was much further down in the file). Eventually I had to give up on using the viewport relative adornment layer because there was no way around these issues!

The approach that finally proved successful was to use an owner controlled adornment layer and position the annotations within vsInk (i.e. not using any of Visual Studio's automatic positioning). This approach resolved all of the problems mentioned above as Visual Studio always reported the correct information for owner controlled adornment layers. While this solved the problem of incorrect values it increased the challenge of actually positioning the annotations. The final solution involved using tracking points in Visual Studio. However this did have the benefit of handling code changes without much additional work (just needed to add the extra events and handle deletions). There is one potential limitation with this approach – it currently assumes all the lines are the same height while Visual Studio allows for lines to be different heights. If this happens then the annotations are going to be incorrectly positioned.

The usability study identified some outstanding challenges with combining annotations and code. One challenge is how to group strokes together into annotations – which in turn causes problems with the repositioning of strokes. I tried two approaches to handling this challenge but neither was entirely successful. The second challenge is how to handle tall annotations without them disappearing unexpectedly.

Previous research has shown that grouping strokes is not easy (Shilman, Simard et al. 2003; Shilman and Wei 2004; Wang, Shilman et al. 2006). Part of the challenge is the huge variety of different types of annotations, both from the same person and between people (Marshall 1997). An ideal system would need to handle this huge range and do it in a way that would improve over time.

Annotations in vsInk are built as the strokes are added or removed – strokes are only added to a single annotation. This could potentially be one reason why the grouping is inaccurate – people do not always add strokes to an annotation in order. Previous research has tried to group annotations by processing all strokes whenever a stroke is added or deleted (Shilman and Viola 2004; Wang, Shilman et al. 2006). While this can give more accurate grouping results it needs to be fast – especially as the amount of work needed increases exponentially with the number of strokes.

Another limitation is vsInk does not use any contextual information. The only contextual information used in deciding to group strokes is the location of the other annotations. Program code itself is a rich source of contextual information that can potentially be used to enhance grouping. For example, when a person is underlining they tend to stay reasonably close to the bottom of the text (Golovchinsky and Denoue 2002). If they then start a new underline on the next line of code it is most likely to be a new annotation, not an extension of an existing one. The same applies for higher levels of code structure – e.g. a stroke in a method is more likely to belong to an annotation within the same method than outside.

The other main challenge identified in the usability is tall annotations tended to disappear. This is caused by vsInk using a single anchor point for each annotation. While this is acceptable for short annotations it fails as annotations increase in height. While the anchor for an annotation is not visible vsInk hides the entire annotation. While this did not happen very often (only 4 annotations out of the 194 had this problem) it does happen in a specific scenario – using arrows to indicate code should be moved to another location. This then caused confusion when the user was following the arrow to see where it went.

One solution mentioned in previous research is to break a tall annotation into shorter segments (Golovchinsky and Denoue 2002) with each segment having its own anchor. This approach was considered but a trial implementation uncovered an interesting flaw. If the user made a change within an annotation the anchor points would become invalid. If a line (or lines) is inserted a gap appeared in the annotation and if a line (or lines) is deleted parts of the annotation would also be deleted. (Golovchinsky and Denoue only looked at read-only documents; the only changes allowed

were font or page size, so they did not have this issue.) While it is potentially possible to handle this issue it raises more challenges due to issues added by the “fixes”. Therefore the tall annotation problem has been left unresolved.

In addition to the two main challenges a number of other interesting points were raised during the usability testing. One of the interesting suggestions during the user study is to include a way to zoom the interface during inking as it is difficult to accurately draw ink on the code. Agrawala and Shilman (2005) provided an implementation of zooming called DIZI to try to handle this issue. DIZI provided a pop-up zoom region that automatically moved as the user annotated. When the user finished annotating the annotation would be scaled back to match the size of the original. A user study found the zooming was most useful when there was limited whitespace. This may be useful for annotating code, especially for dense “blocks” of code.

Another suggestion is to include ink actions in the undo history within Visual Studio. People are no more accurate using digital ink than with a keyboard and mouse and still tend to make mistakes. One common mistake that was observed is a person would add a new stroke that was incorrectly grouped, so they would need to change to the eraser to remove the stroke and try again. A much faster (and more accurate) approach would be to push the undo button. Visual Studio does expose the undo history API, making it reasonably easy to extend it. The main challenge is modifying vsInk so it correctly works (e.g. handling grouping, adornment updates, etc.) Related to implementing undo is other common functionality exposed by modern applications including cut/copy/paste. Again the challenge is not Visual Studio but how the functionality should work within vsInk.

A final suggestion that would be worthwhile is including the underlying code in the various navigation elements. All of the full time developers made this suggestion, especially after the code had been edited. One of these developers commented it was hard to see why they had added the annotation without the code as context.

7 Future Work

Some of the possibilities for enhancing vsInk in future include:

- Investigating ways to improve grouping, including: modifying the rules in the grouping algorithm, including contextual data in the grouping process, attempting to group all strokes in a single pass (rather than just when the stroke is added) and using machine learning techniques to "learn" a grouping algorithm.
- Splitting annotations into sections with individual anchors for each section. This would solve the tall annotation issue and allow annotations to fold when a region is collapsed or expanded. This would need some way to handles changes to the underlying code (e.g. if lines are added or deleted).
- Investigating how annotations should change if the underlying code changes. This is more than just handling tall annotations with multiple sections, it would include horizontal changes to the code (e.g. if some text is underlined and a new word is added, what should happen to the annotation?)
- Adding addition integration with Visual Studio: undo history, cut/copy/paste, etc. Also included would be displaying code in the navigation elements.
- Expanding navigation options. The current navigation is constrained to a single file: it would be good to add additional navigation features across multiple files. Possibilities include adding a timeline to allow users to retrace their annotation sequence, a "clippings" book showing all the annotations in a project, etc.
- Implementing zooming to help with the annotations. This could be a static zoom that the user chooses or some form of dynamic zooming (similar to DIZI (Agrawala and Shilman 2005)). An interesting option would to be to use the adjustable line heights provided by Visual Studio – the positioning algorithm would need changes to handle this.

There is also a need for additional user studies to understand why and how people annotate code. This project is based on the assumption that annotating code within an IDE is useful but there is no empirical evidence for this. If annotating code is useful, what scenarios would it be useful for? Previous studies have looked at using annotations for code reviews (Priest and Plimmer 2006), marking (Plimmer and Mason 2006; Plimmer 2010), presenting lectures (Anderson, McDowell et al. 2005; Anderson, Davis et al. 2007) – these may also be valid areas for IDE-based annotations.

8 Conclusions

vsInk is a tool for annotating program code using the Visual Studio code editor. A user can add digital ink directly in the editor and still use all the functionality of the editor. vsInk resolves the issue of integrating ink and code together where other attempts had failed. While vsInk is more successful than previous attempts it still has some limitations. The two main limitations are grouping strokes into annotations in a more natural way and handling tall annotations.

Functional deficiencies in earlier prototypes prevented additional research into the value of annotating program code with digital ink. Using vsInk we can now investigate this area. We can also investigate how ink should behave when the associated text in a document changes.

9 Appendix

9.1 Usability Test – Protocol

Before Testing

Step	Instruction
1.	Print out: • Participant information sheet • Participant instructions • Questionnaire
2.	Ensure vsInk is installed and runs
3.	Ensure AutoUpdate is installed and runs
4.	Ensure Morae recorder works
5.	Copy fresh copy of Demo code

During Testing

Step	Instruction
1.	Welcome subject
2.	Explain project and purpose of test – hand out participant information sheet
3.	Sign ethics approval
4.	Pre-test questionnaire
5.	Show vsInk – explain main functionality
6.	Provide 3 minutes familiarisation time – allow subject to play around with vsInk
7.	Hand out participant instructions and allow them to read it, check they understand everything
8.	Open Demo project and start Morae recording
9.	Perform scenario one (8 minutes)
10.	Mid-test questionnaire
11.	Run AutoUpdate
12.	Perform scenario two (8 minutes)
13.	Post-test questionnaire
14.	Review positions of moved annotations and ask whether subject thinks it is correct
15.	Stop Morae recording
16.	Informal interview – ask their opinion of the vsInk (good, bad, improvements)
17.	Thank them for their time

After Testing

Step	Instruction
1.	Collect all paperwork
2.	Save Morae recording and back-up

9.2 Usability Test – Participant Instructions

Background

You are an engineer at one of Elbonia's largest companies – Elbonian Exports Ltd. Since you are a recent hire your job is to review all the code written by the other engineers to learn how things work. Alice, one of the other engineers, is trying to improve the quality of code written in your team. So she has tasked you to find all the errors in the code at the same time you are reading. Your job is not to fix the errors – that is too important a job for an intern to do – merely to find them. So after you have found any errors Alice will send them back to the original engineer for them to fix. Unfortunately that also means you will have to review the code again afterwards to ensure that they did actually fix the errors!

Scenario One: Initial Review

You have been given some code to review by Alice for the new auto-sorter that is coming. The code was written by an engineer called Wally, who isn't the best of coders. Wally's job is to write the controller application – somebody else will write the user interface and somebody else the interface with the actual auto-sorter.

Task: Review the code to ensure it meets the Elbonian Exports Ltd. coding guidelines (see Guidelines.) While reviewing the code annotate any violations found so Wally can correct them.

Scenario Two: Second Review

After you reviewed the code Alice sent it to Wally to fix. He has gone through your annotations and says he has fixed all the important issues. Alice would like you to review the code again, comparing your annotations to what Wally has fixed.

Task: Review the code again to see that Wally has fixed all the violations that you found. Mark any violations that you find in a different colour so Wally knows what he missed the first time.

Guidelines

1. The following items should use Pascal Casing:
   - Classes (e.g. AppDomain)
   - Interfaces (e.g. IBusinessService)
   - Methods (e.g. ToString)
   - Properties (e.g. BackColour)
   - Namespaces (e.g. System.Drawing)
2. The following items should use Camel Casing:
   - Private fields (e.g. listItem)
   - Variables (e.g. listOfValues)
   - Parameters (e.g. typeName)
3. Underscores should not be used in any item names (e.g. _myField)
4. All interface names should be prefixed with a capital I (e.g. IBusinessService)
5. All methods and properties should have document comments

```
/// <summary>
/// An example set of doc-comments.
/// </summary>
```

6. Members in classes and interfaces should be in the following order:
   i. Private fields
   ii. Constructors
   iii. Properties
   iv. Public methods
   v. Private methods
7. Use C# type aliases instead of types from the System namespace (e.g. int instead of Int32)
8. Each class or interface must be in its own file

9.3 Usability Test – Questionnaire

Pre-Test Survey

	Always				Never
	1	2	3	4	5
1. I use Visual Studio 2010 for development					
2. I use Visual Studio 2010 for reviewing code					
3. I use digital annotation tools for reviewing documents					
4. I have used pen input on a computer					
5. I review and write annotations on my own code					
6. I review and write annotations on other people's code					

POST SCENARIO 1

	Agree				Disagree
	1	2	3	4	5
1. The exercise was enjoyable					
2. I understand the task					
3. The interaction helped with my task completion					
4. Annotating code within Visual Studio was easy					

POST SCENARIO 2

	Agree				Disagree
	1	2	3	4	5
1. The exercise was enjoyable					
2. I understand the task					
3. The interaction helped with my task completion					
4. Annotating code within Visual Studio was easy					
5. Finding previous annotations in Visual Studio was easy					
6. The annotations in Visual Studio moved in an understandable way when the code was annotated					
7. I would like to use digital ink in Visual Studio in the future					
8. Using digital ink annotations in Visual Studio is better than my current practise for annotating code					

SYSTEM USABILITY SURVEY

	Agree				Disagree
	1	2	3	4	5
1. I think that I would like to use this system frequently					
2. I found the system unnecessarily complex					
3. I thought that the system was easy to use					
4. I think that I would need the support of a technical person to be able to use this system					
5. I found the various functions in this system were well integrated					
6. I thought there was too much inconsistency in this system					
7. I would imagine that most people would learn to use this system very quickly					
8. I found the system very cumbersome to use					
9. I felt very confident using the system					
10. I needed to learn a lot of things before I could get going with this system					

10 References

Agosti, M., G. Bonfiglio-Dosio, et al. (2007). "A historical and contemporary study on annotations to derive key features for systems design." International Journal on Digital Libraries **8**: 1-19.

Agrawala, M. and M. Shilman (2005). DIZI: A Digital Ink Zooming Interface for Document Annotation Human-Computer Interaction - INTERACT 2005, Springer Berlin / Heidelberg.

Anderson, R., R. Anderson, et al. (2004). Experiences with a tablet PC based lecture presentation system in computer science courses. Proceedings of the 35th SIGCSE technical symposium on Computer science education. Norfolk, Virginia, USA, ACM**:** 56-60.

Anderson, R., P. Davis, et al. (2007). "Classroom Presenter: Enhancing Interactive Education with Digital Ink." Computer **40**(9): 56-61.

Anderson, R., L. McDowell, et al. (2005). Use of classroom presenter in engineering courses. Frontiers in Education, 2005. FIE '05. Proceedings 35th Annual Conference.

Bangor, A., P. T. Kortum, et al. (2008). "An empirical evaluation of the System Usability Scale." International Journal of Human-Computer Interaction **24**(6): 574-594.

Bargeron, D. and T. Moscovich (2003). Reflowing digital ink annotations. Proceedings of the conference on Human factors in computing systems - CHI '03. New York, New York, USA, ACM Press**:** 385-392.

Bradley, A. (2012). "Collage Framework and Code Markup Tools."

Brandl, P., C. Richter, et al. (2010). NiCEBook: supporting natural note taking. Proceedings of the 28th international conference on Human factors in computing systems - CHI '10. New York, New York, USA, ACM Press**:** 599-608.

Brooke, J. (1996). SUS: A “quick and dirty” usability scale. Usability Evaluation in Industry. P. W. Jordan, B. Thomas, B. A. Weerdmeester and McClelland. London, UK, Taylor & Francis**:** 189-194.

Brush, A. J. B., D. Bargeron, et al. (2001). Robust annotation positioning in digital documents. Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '01. New York, New York, USA, ACM Press**:** 285-292.

Chang, S. H.-H., R. Blagojevic, et al. (2012). "RATA.Gesture: A Gesture Recognizer Developed using Data Mining." Artificial Intelligence for Engineering Design, Analysis and Manufacturing **23**(3): 351-366.

Chang, S. H.-H., X. Chen, et al. (2008). Issues of extending the user interface of integrated development environments. Proceedings of the 9th ACM SIGCHI New Zealand Chapter's International Conference on Human-Computer Interaction Design Centered HCI - CHINZ '08. New York, New York, USA, ACM Press**:** 23-30.

Chatti, M. A., T. Sodhi, et al. (2006). u-Annotate: An Application for User-Driven Freeform Digital Ink Annotation of E-Learning Content. International Conference on Advanced Learning Technologies - ICALT. Kerkrade, The Netherlands, IEEE Computer Society**:** 1039-1043.

Chen, X. and B. Plimmer (2007). CodeAnnotator. Proceedings of the 2007 conference of the computer-human interaction special interest group (CHISIG) of Australia on Computer-human interaction: design: activities, artifacts and environments - OZCHI '07. New York, New York, USA, ACM Press**:** 211.

Golovchinsky, G. and L. Denoue (2002). Moving markup. Proceedings of the 15th annual ACM symposium on User interface software and technology - UIST '02. New York, New York, USA, ACM Press**:** 21.

Guyon, I., L. Schomaker, et al. (1994). UNIPEN project of on-line data exchange and recognizer benchmarks. Computer Vision & Image Processing., Proceedings of the 12th IAPR International. Conference on.

Liao, C., F. Guimbretière, et al. (2005). PapierCraft: A System for Interactive Paper. Proceedings of the 18th annual ACM symposium on User interface software and technology - UIST '05. New York, New York, USA, ACM Press: 241-244.

Lichtschlag, L. and J. Borchers (2010). CodeGraffiti: Communication by Sketching for Pair Programming. Adjunct proceedings of the 23nd annual ACM symposium on User interface software and technology - UIST '10. New York, New York, USA, ACM Press: 439-440.

Marshall, C. C. (1997). Annotation: from paper books to the digital library. Proceedings of the second ACM international conference on Digital libraries - DL '97. New York, New York, USA, ACM Press: 131-140.

Marshall, C. C. and A. J. B. Brush (2004). Exploring the relationship between personal and public annotations. Proceedings of the 2004 joint ACM/IEEE conference on Digital libraries - JCDL '04. New York, New York, USA, ACM Press: 349.

Marshall, C. C., M. N. Price, et al. (1999). Introducing a digital library reading appliance into a reading group. Proceedings of the fourth ACM conference on Digital libraries. Berkeley, California, United States, ACM: 77-84.

Marshall, C. C., M. N. Price, et al. (2001). Designing e-books for legal research. Proceedings of the first ACM/IEEE-CS joint conference on Digital libraries - JCDL '01. New York, New York, USA, ACM Press: 41-48.

Microsoft. "Inside the Editor." Retrieved 1-Aug-2012, 2012, from http://msdn.microsoft.com/en-us/library/dd885240.aspx.

Microsoft (2007) "Ink Serialized Format Specification." **2012**.

Morris, M. R., A. J. B. Brush, et al. (2007). Reading Revisited : Evaluating the Usability of Digital Display Surfaces for Active Reading Tasks. Workshop on Horizontal Interactive Human-Computer Systems - TABLETOP. Newport, Rhode Island, IEEE Comput. Soc: 79-86.

O'Hara, K. and A. Sellen (1997). A comparison of reading paper and on-line documents. Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '97. New York, New York, USA, ACM Press: 335-342.

Olsen, D. R., T. Taufer, et al. (2004). ScreenCrayons: annotating anything. Proceedings of the 17th annual ACM symposium on User interface software and technology - UIST '04. New York, New York, USA, ACM Press: 165-174.

Phelps, T. A. and R. Wilensky (2000). "Robust intra-document locations." Computer Networks **33**: 105-118.

Plimmer, B. (2010). A comparative evaluation of annotation software for grading programming assignments. Proceedings of the Eleventh Australasian Conference on User Interface - Volume 106. Brisbane, Australia, Australian Computer Society, Inc.: 14-22.

Plimmer, B., S. H.-H. Chang, et al. (2010). iAnnotate: exploring multi-user ink annotation in web browsers. AUIC '10 Proceedings of the Eleventh Australasian Conference on User Interface. Darlinghurst, Australia, Australian Computer Society, Inc: 52-60.

Plimmer, B. and P. Mason (2006). A pen-based paperless environment for annotating and marking student assignments. Australasian User Interface Conference - AUIC. Hobart, Tasmania, Australia, Australian Computer Society, Inc: 37-44.

Price, M. N., G. Golovchinsky, et al. (1998). Linking by inking: trailblazing in a paper-like hypertext. Proceedings of the ninth ACM conference on Hypertext and hypermedia : links, objects, time and space---structure in hypermedia systems links, objects, time and space---structure in hypermedia systems - HYPERTEXT '98. New York, New York, USA, ACM Press: 30-39.

Priest, R. and B. Plimmer (2006). RCA: experiences with an IDE annotation tool. Proceedings of the 6th ACM SIGCHI New Zealand chapter's international conference on Computer-human interaction design centered HCI - CHINZ '06. New York, New York, USA, ACM Press: 53-60.

Ramachandran, S. and R. Kashi (2003). An architecture for ink annotations on Web documents. Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings. Edinburgh, Scotland, UK, IEEE Comput. Soc. **1:** 256-260.

Schilit, B. N., G. Golovchinsky, et al. (1998). Beyond paper: supporting active reading with free form digital ink annotations. Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '98. New York, New York, USA, ACM Press**:** 249-256.

Sellen, A. and R. Harper (2003). "The Myth of the Paperless Office."

Shilman, M., P. Simard, et al. (2003). Discerning structure from freeform handwritten notes. Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings. Edinburgh, Scotland, UK, IEEE Comput. Soc. **1:** 60-65.

Shilman, M. and P. Viola (2004). Spatial Recognition and Grouping of Text and Graphics. Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBIM). Grenoble, France**:** 91-95.

Shilman, M. and Z. Wei (2004). Recognizing Freeform Digital Ink Annotations. DOCUMENT ANALYSIS SYSTEMS VI. S. Marinai and A. R. Dengel. Berlin, Heidelberg, Springer Berlin Heidelberg. **3163:** 107-110.

Steimle, J., O. Brdiczka, et al. (2008). CoScribe: Using Paper for Collaborative Annotations in Lectures. 2008 Eighth IEEE International Conference on Advanced Learning Technologies. Santander, Cantabria, Ieee**:** 306-310.

Sutherland, I. (1964). Sketch pad a man-machine graphical communication system. Proceedings of the SHARE design automation workshop. **23:** 329-346.

Tashman, C. S. and W. K. Edwards (2011). Active reading and its discontents. Proceedings of the 2011 annual conference on Human factors in computing systems - CHI '11. New York, New York, USA, ACM Press**:** 2927.

Tashman, C. S. and W. K. Edwards (2011). LiquidText. Proceedings of the 2011 annual conference on Human factors in computing systems - CHI '11, ACM Press.

Wang, X. and S. Raghupathy (2007). Ink Annotations and their Anchoring in Heterogeneous Digital Documents. Analysis. Curitiba, Brazil**:** 163-167.

Wang, X., M. Shilman, et al. (2006). Parsing Ink Annotations on Heterogeneous Documents. Sketch Based Interfaces and Modeling - SBIM, Eurographics.

Watt, S. M. and T. Underhill (2011). Ink Markup Language (InkML).

Wilcox, L. D., B. N. Schilit, et al. (1997). Dynomite: a dynamically organized ink and audio notebook. Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '97. New York, New York, USA, ACM Press**:** 186-193.

Wolfe, J. L. (2000). Effects of annotations on student readers and writers. Proceedings of the fifth ACM conference on Digital libraries - DL '00. New York, New York, USA, ACM Press**:** 19-26.

Zyto, S., D. Karger, et al. (2012). Successful classroom deployment of a social document annotation system. Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems - CHI '12, New York, New York, ACM Press.