vsInk – Integrating Digital Ink with Program Code in Visual Studio

Craig J. Sutherland, Beryl Plimmer
Department of Computer Science
University of Auckland

cj.sutherland@auckland.ac.nz, beryl@cs.auckland.ac.nz

Abstract

We present vsInk, a plug-in that affords digital ink annotation in the Visual Studio code editor. Annotations can be added in the same window as the editor and automatically reflow when the underlying code changes. The plug-in uses recognisers built using machine learning to improve the accuracy of the annotation's anchor. The user evaluation shows that the core functionality is sound.

Keywords: Digital ink, code annotation, Visual Studio.

1 Introduction

This paper presents a technical, usable solution for adding digital ink annotation capacity to the code editor of an Integrated Development Environments (IDE). We support a transparent ink canvas on the code editor window; the user can add, move and delete digital ink. The annotations are anchored to a line of the underlying code and maintain their relative position to this line as the window scrolls and lines above are added or removed. The application also handles collapsible regions within the editor and provides spatial and temporal visualizations to support navigation.

Using a pen is a natural and easy way to annotate documents. One of the main reasons why people prefer paper documents to online documents is the ability to annotate easily using just a pen (O'Hara and Sellen, 1997). This form of annotation does not interrupt the reading process and allows the reader the freedom to annotate as they prefer. More recent research by Morris, Brush and Meyers (2007) and Tashman and Edwards (2011) found that pen-based computers allowed readers the same easy ability to annotate as paper and even overcome some of the limitations of paper (e.g. re-finding existing annotations and lack of space for annotations).

One form of document that is often reviewed is program code (Priest and Plimmer, 2006). While program code is a form of text document it differs significantly from other documents. Code is predominately non-linear – it is broken up into methods and classes – and is often split across multiple files. It can be printed out and annotated but as the program size increases it becomes more difficult to follow the flow of program logic. To help developers read and understand code they usually use tools like IDEs.

Copyright © 2013, Australian Computer Society, Inc. This paper appeared at the 14th Australasian User Interface Conference (AUIC 2013), Adelaide, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 139. Ross T. Smith and Burkhard Wuensche, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

An IDE is a complex environment that provides many tools for working with program code. Tools include editors with intelligent tooltips and instant syntax feedback, powerful debuggers for tracing program flow and examining state and different visualisers for showing how the code fits together. Text comments interspersed with the code are widely used for documentation and notes. For example, a developer may add a TODO comment to mark something that needs to be done.

One feature that is lacking is the ability to use digital ink for annotations. Ink annotations are a different level of visualization. They are spatially linked to specific parts of the document but they are quickly and easily discernible from the underlying document (O'Hara and Sellen, 1997).

Previous prototypes for digital ink annotation in IDEs (Chen and Plimmer, 2007, Priest and Plimmer, 2006) failed to support a canvas on the active code window because of the lack of appropriate extension points in the APIs of the IDEs (Chang et al., 2008). These earlier prototypes cloned the code into a separated annotation window resulting in an awkward user experience.

We have built a plug-in to Visual Studio 2010, called vsInk *that* allows people to use digital ink to annotate code in the active code editor. The requirements for vsInk were garnered from the literature on IDE annotation and general document annotation. In order to achieve the desired functionality a custom *adornment* layer is added to the code window. vsInk applies digital ink recognition techniques to anchor and group the annotations to the under lying code lines. We ran a two-phase user evaluation to ensure that the basic usability was sound and identify where more development is required.

2 Related Work

The literature reports three attempts to integrate digital annotations in IDEs. Priest and Plimmer (2006) attempted to modify Visual Studio 2005. With their plug-in, called RichCodeAnnotator (RCA), they tried but failed to add annotations directly to the code editor. Instead RCA used a custom window that allowed users to annotate over a copy of the code. The plug-in grouped strokes together into annotations based on simple spatial and temporal rules. The first stroke of an annotation is called the linker and is used to generate the anchor for the whole annotation. RCA allowed for two types of linkers – circles and lines. Simple heuristic rules were used to determine the linker type. When code lines are added or removed above the anchor the whole annotation is moved. The next IDE to be modified was Eclipse with CodeAnnotator (Chen and Plimmer, 2007). This plug-in was based on the experiences with RCA and used the same approach (e.g. grouping strokes and using a linker).

One issue with both plug-ins was integrating the annotation surface directly into the code editor (Chang et al., 2008). Neither Visual Studio 2005 nor Eclipse provided sufficient extensibility hooks to allow third parties to directly integrate with the editor. To get around this issue both RCA and CodeAnnotator used a separate window for annotating. The code in the annotation window is read-only although it is automatically refreshed when the code is modified.

Another common issue is how to group together individual strokes to compose annotations. Both RCA and CodeAnnotator used simple rules for determining whether a stroke belonged to an existing annotation. These rules were based partially on previous work with digital annotations (Golovchinsky and Denoue, 2002). Both plug-ins used two simple rules – a stroke was added to an existing annotation if it was added within two seconds of the last stroke or it was within a certain distance of other strokes. The area assigned to the annotation is indicated visually with a box that expands automatically as the annotation grows. There is the implication that they were only semi-effective as both plug-ins made adjustments to the grouping process. RCA allowed users to manually select an existing annotation to force strokes to join. CodeAnnotator changed the rules defining whether a stroke was close to an existing stroke. The final rule involved different distances for each side. Neither paper reports the accuracy of their grouping strategy.

Another common issue with annotations is how to anchor them to the underlying text. This is a common issue with annotations in general, not just for program code. Using an x-y co-ordinate for the anchor works fine for static documents but as soon as the document can be modified the x-y co-ordinate becomes meaningless (Brush et al., 2001, Golovchinsky and Denoue, 2002, Bargeron and Moscovich, 2003). Previous attempts at solving this issue have typically included context from the underlying document to generate the anchor point. For example, XLibris (Golovchinsky and Denoue, 2002) uses the underlying text while u-Annotate (Chatti et al., 2006)and iAnnotate (Plimmer et al., 2010) both use HTML DOM elements. The approached used by RCA and CodeAnnotator is to select the closest line of code to the annotation. They do not mention how this line is tracked as the code changes.

One issue with annotations in an IDE is how to navigate between them. This is particularly problematic for program code because of the non-linear flow of the documents. Navigation was partially addressed in CodeAnnotator by the addition of an outline window (Chen and Plimmer, 2007). The navigation window displayed a thumbnail of each annotation in the document. Selecting an annotation automatically scrolls to the annotation. The main limitation of this is it assumes that the user is only interested in navigating annotations in the same order they are in the document. Given the non-linear nature of code users are likely to add comments as they trace through the code. Indeed other annotations systems such as Dynomite (Wilcox et al., 1997) and XLibris (Schilit et al., 1998) provide timeline views that organise annotations in the order they were added.

The final IDE plug-in reported in the literature is CodeGraffiti (Lichtschlag and Borchers, 2010). CodeGraffiti was designed as a pair programming tool that extends the Xcode IDE. One person would use CodeGraffiti in Xcode and a second person can view and add annotations via a remote session (e.g. on an iPad or a second computer). There are few details provided about CodeGraffiti's functionality but it appears that it works by anchoring annotations to lines of code. It does not mention whether annotations are allowed directly in the editor, how strokes are grouped or any navigation support.

One issue that has not been mentioned in any study is how annotations should behave when the underlying code is hidden. Most IDEs allow users to define collapsible regions within files which can be collapsed and expanded as desired. Brush, *et al.* (2001) investigated how people expected annotations to behave when the underlying context was changed or deleted but this assumes permanent changes to the document not temporary changes like collapsing a region.

In summary, the current literature describes a number of issues in adding annotations to IDEs. Limitations in the IDE extensibility models have prevented past attempts from integrating ink directly into the code editor window. Other issues include how to group together single strokes into annotations, how to calculate an anchor point for repositioning annotations and how to navigate through existing annotations. One area that has not been investigated at all is handling collapsible regions within code.

3 Requirements

From the literature review five requirements were identified for vsInk. First, annotations need to be directly integrated within the code editor. Second, strokes need to be automatically grouped together into annotations. Third, annotations need to be anchored to the underlying code in a way that allows them to be consistently repositioned after any modification to the code. Fourth, support is needed for collapsible regions. And fifth, it should be easy for users to navigate through annotations.

3.1 Direct Editor Integration

Users should be able to directly add annotations within the code editor. As mentioned above previous attempts required the user to annotate code in a separate read-only window (Chang et al., 2008) which has the potential to cause confusion. Adding an annotation should be as simple as turning on ink mode and drawing. None of the existing editor functionality should be lost (e.g. the user should still be able to modify the code, debug, etc.).

3.2 Grouping Strokes into Annotations

As users add strokes they should be grouped together in a way that appears natural. The rules used in RCA and CodeAnnotator (Chen and Plimmer, 2007, Priest and Plimmer, 2006) can be used as a starting point but these may need to be expanded. As an example, when a user is writing text they typically expect all the letters in a word to be grouped together in a single annotation. In contrast annotations on consecutive lines may not belong together.

3.3 Anchoring and Repositioning Annotations

When the code editor is scrolled or the code is modified the annotations should stay positioned relative to the associated code. To handle this introduces two requirements. First some way of identifying an anchor point is needed. vsInk will extend the concept introduced in RCA of the linking stroke (Priest and Plimmer, 2006). To allow for a greater variety of linking strokes the first stroke in an annotation should be classified and the anchor point determined from the stroke type. This anchor point should then be associated with the closest line of code.

Second, when the code is modified within the editor the annotations should move relative to the associated line of code. Effectively this means if lines are added or removed above the annotation the annotation should move down or up.

3.4 Collapsible Region Support

Visual Studio allows a developer to mark up code as belonging to a collapsible region. This involves adding start and end region tags to the program code which are recognised by the editor. The user can toggle these regions by clicking on an indicator in the margin of the editor (see Figure 1).

```
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 – the region is expanded. Bottom view – the same region when collapsed.

When the user collapses a region all the annotations with anchor points inside the region should be hidden. When a region is expanded all the annotations should be restored to their previous positions.

3.5 Navigation

The user should be able to see all the annotations they have added within a file. This requires two types of navigation elements. First there should be an indicator to show a collapsed region contains annotations. Second there should be an overview of all the annotations within the file. The overview should display a view of what the annotation looks like. Also it should allow sorting by either physical position within the file or when the annotation was added.

Both navigation elements should allow the user to navigate to a selected annotation. When a user selects an annotation the editor should automatically scroll to the location in the code and any collapsed regions expanded so the annotation is visible.

4 Implementation

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

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

4.1 Editor Integration

In Visual Studio 2010 it is possible to extend the editor by adding adornments in the editor. A Visual Studio adornment is a graphic element that is displayed over the code in the code editor. A plug-in extends the editor by defining an adornment layer and adding adornments to it. This adornment layer is added to the editor. Visual Studio offers three types of adornment layers – text-relative (associated with the text), view-relative (associated with the viewport) and owner-controlled (Microsoft). vsInk works by adding a new owner-controlled adornment layer. This layer contains an ink canvas that covers the entire viewpoint.

Initially we tried to use the text-relative and viewport-relative layers but both of these resulted in problems. The text-relative layer approach failed because it required each annotation to be associated with a location in the text. vsInk requires a single ink canvas to cover the entire viewport (rather than an adornment per annotation) so that free form inking is supported anywhere in the document.

The viewport-relative layer initially seemed more promising as it allowed the annotations to scroll in sync with the code. However there were a number of scenarios where the scrolling broke (e.g. when moving to the top or bottom of a long file). These appeared to be 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.

Using an owner-controlled adornment layer gives vsInk full control over how the elements are displayed – Visual Studio does not attempt to do any positioning. This flexibility does come at a cost: vsInk now needs to position all the adornments itself. The ink canvas in vsInk is the only UI element that is added directly to the adornment layer – all other UI elements are added to the ink canvas. The ink canvas is positioned so it covers the entire viewpoint – from the top-left corner to the bottom-right. The actual viewport in Visual Studio is more than just the viewable screen area; it also includes some lines above or below the viewable space and is as wide as the widest line.

The annotation anchor requires two items: the line number ($Line_{\#}$) and a line offset ($Offset_{Line}$). The editor in

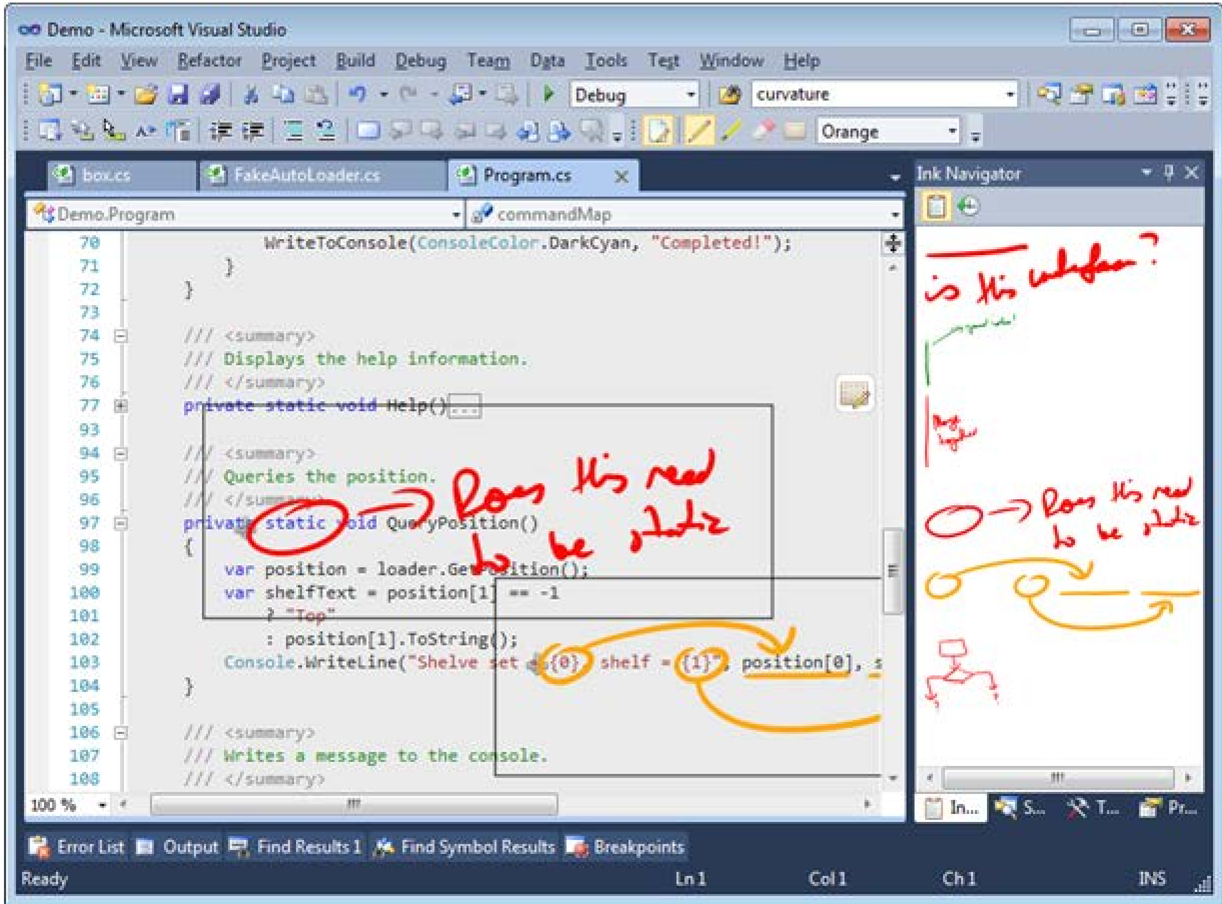

Figure 2: vsInk in Visual Studio 2010.

Visual Studio is broken up into a number of lines (see Figure 3). When a new annotation is started the closest line to the linker anchor point is selected (see Figure 4) – this is $Line_{\#}$. Internally $Line_{\#}$ is recorded as a Visual Studio tracking point. Storing $Line_{\#}$ as a tracking point enables vsInk to use Visual Studio's automatic line tracking to handle any changes to the code.

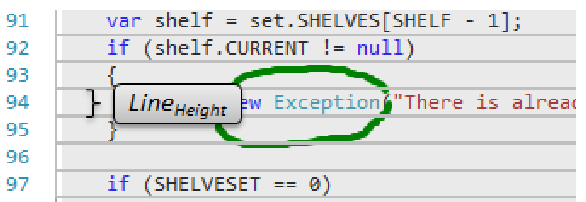

Figure 3: The editing surface in Visual Studio with the individual lines marked.

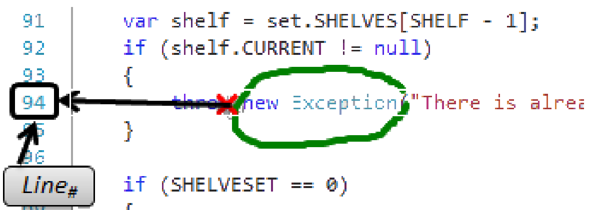

Figure 4: The process of selecting the closest line for $Line_{\#}$.

When the user scrolls through a document Visual Studio fires an event notifying any listeners that the viewport has changed (this change can be either repositioning or regeneration). vsInk listens for this event and updates every annotation on the canvas when it is received. First, each element is checked whether its $Line_{\#}$ is visible If $Line_{\#}$ is not visible then the annotation is hidden. If $Line_{\#}$ is visible it is checked to see if it is inside a collapsed region, again the annotation is hidden if this check is true. If both of these checks pass the annotation is displayed. Finally a translation transform is applied to each annotation to move it into the correct position.

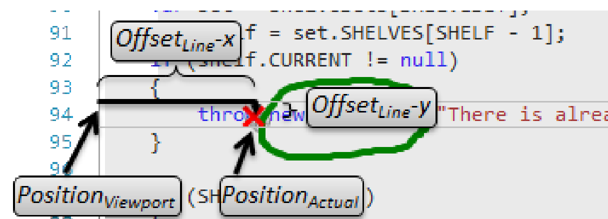

Figure 5: The calculation from $Position_{ViewPort}$ to $Position_{Actual}$.

The actual positioning of each annotation requires a number of 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 6). Finally $Offset_{Line}$ is added to $Position_{ViewPort}$ to get the actual position ($Position_{Actual}$) (see **Figure 5**).

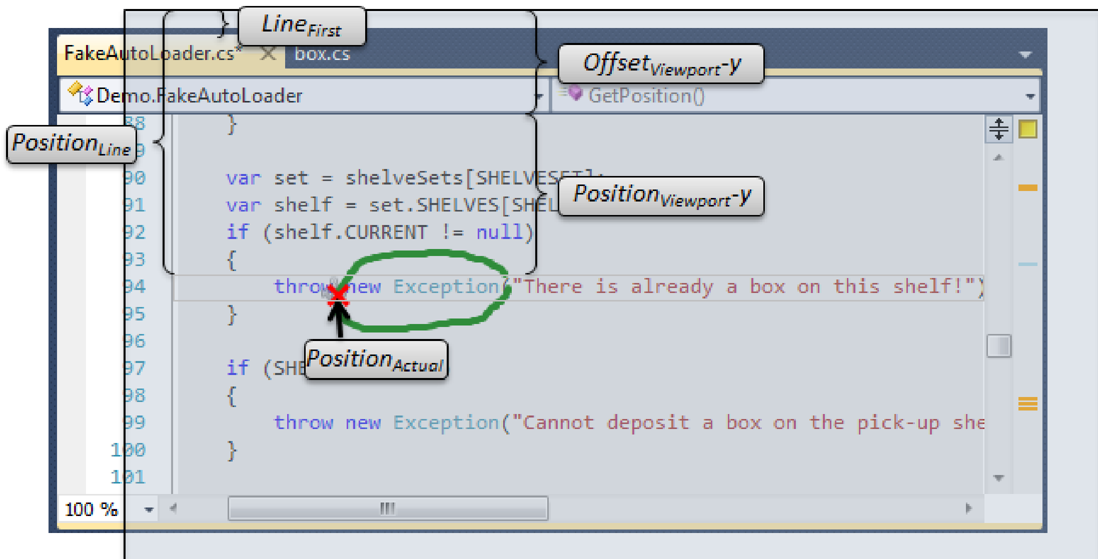

Figure 6: The Visual Studio editor. The grey box shows the viewport region used by Visual Studio.

$Position_{Actual}$ is used to translate the annotation into its correct position on the canvas.

Collapsible region support is added by listening to the relevant events in the editor. The region manager in Visual Studio has three events for handling changes to regions (*RegionsChanged*, *RegionsCollapsed* and *RegionsExpanded*). When any of these events are received vsInk performs an update of the ink canvas (the same as when the viewport changes). Since a collapsed region check is already performed in the update no further changes were needed to support collapsible regions.

Changes to the code are handled in a similar way. vsInk listens to the events that fire whenever the underlying code is changed and performs a canvas update. Because a tracking position is used to record the closest line the line number is automatically updated when lines are added or removed. The final step is to detect whenever a line with an annotation anchor has been deleted. In this case the entire annotation is deleted as well.

The annotation ink and associated elements are serialised to a binary format and saved automatically every time the associated code file is saved. The strokes are stored using Microsoft's Ink Serialise Format (ISF). When a new document window is opened vsInk checks to see if there is an associated ink file. If the file exists the existing annotations are deserialised and added to the canvas, otherwise a new blank canvas is started.

4.2 Grouping Strokes into Annotations

Grouping annotations is performed by using simple rules. The initial version of vsInk used a boundary check to group strokes with annotations. A boundary region for each annotation is calculated by getting the bounding box for the annotation and adding 30 pixels to each side (see Figure 7). vsInk tests a new stroke against all annotations in a file to see if the new stroke intersects any existing boundary region. If a stroke intersects a boundary region for multiple annotations it is added to the first annotation found. If the stroke does not intersect any existing annotations it starts a new annotation.

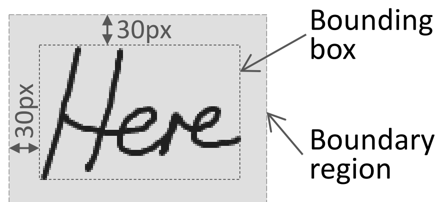

Figure 7: Example of the boundary region for an annotation.

Usability testing (see below) showed that this was not accurate enough. The two main conditions under which this failed were when the user started a new annotation too close to an existing annotation or the user was trying to add a new stroke to an existing annotation but was too far away. In addition when multiple annotations were found the new stroke was often added to the wrong annotation!

Three changes were made to address these issues. First, the boundary was decreased to 20 pixels. Second, a timing check was added – if a new stroke was added within 0.5 seconds of the last stroke it was added to the same annotation. 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 final change was for selecting which annotation when multiple possible annotations were found. The annotation chosen is the annotation that has the closest middle point to the starting point of the stroke. Euclidean distances are used to calculate the closest middle point.

4.3 Anchoring Annotations

vsInk adopts the concept of a linking stroke for generating the anchor from Priest and Plimmer (2006). In vsInk the linking stroke is the first stroke of a new annotation. The actual anchor point is calculated based on

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

Table 1: Recognised linker types. The red cross indicates the location of the anchor.

the type of the first stroke. Both RCA and CodeAnnotator used some simple heuristics for determining the type of the linking stroke which could be a line or a circle (Chen and Plimmer, 2007, Priest and Plimmer, 2006) but this was too limiting, especially as new linker types are needed.

To overcome this we used Rata.Gesture (Chang et al., 2012) to recognise the stroke type. Rata.Gesture is a tool that was developed at the University of Auckland for generating ink recognisers. Rata works by extracting 115 features for each stroke and then training a model to classify strokes. This model is then used in the recogniser for classifying strokes.

To generate the recogniser for vsInk an informal user survey was performed to see what the most common types of linking strokes would be. This produced the list of strokes in Table 1. Ten users were then asked to provide ten examples of each stroke, giving a total of 600 strokes to use in training. These strokes were manually labelled and Rata used to generate the recogniser.

When a new annotation is started the recogniser is used to classify the type of linker. Each linker type has a specific anchor location (see Table 1) – this location is used to find the $Line_{\#}$ for anchoring.

4.4 Annotation Adornments

Each annotation can have a number of associated adornments. These adornments are slightly different from the Visual Studio adornments in two ways: they are associated with an annotation rather than an adornment layer and their visibility is controlled by vsInk. There are two default adornments in vsInk: the boundary region indicator and the anchor indicator. In addition vsInk allows for third parties to write their own custom adornments. An example of a custom adornment is provided in the project; it displays the user name of the person who added the annotation.

When an annotation is added a factory class for each adornment is called to generate the adornments for the new annotation. This process is called for both loading annotations (e.g. when a document is opened) and for a user adding a new annotation. Each adornment is then added to a sub-layer of the ink canvas. The sub-layer is needed to prevent the adornments from being selected and directly modified by the user. Custom adornments can be added to vsInk by adding a new factory class.

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 parts to navigation – collapsed region support and a navigation outline. Collapsed region support adds an icon to a sub-layer of the ink canvas whenever a collapsed region contains annotations. The addition or deletion of the icon is performed during the canvas update process, which is triggered whenever a region is changed. This ensures the icon is always up-to-date and only displayed when there are annotations in a collapsed region.

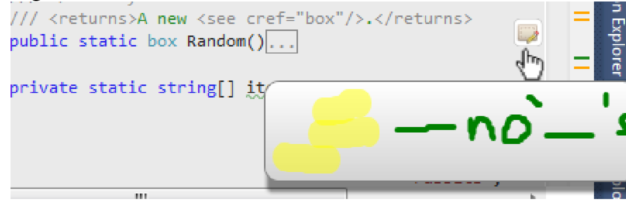

Figure 8: The hidden annotation icon and tooltip preview.

Clicking on the icon automatically expands as many collapsed regions as needed and scrolls the editor so the annotation is in view. In addition the annotation is "flashed" to show the user where the annotation is. The actual "flash" is implemented by the different adornments – the default implementation is to change the border size for the boundary region. In addition when the user hovers the pen (or mouse) over the icon a thumbnail is displayed of the entire annotation (see Figure 8).

The navigation outline is implemented as a separate tool window within Visual Studio. This gives the user full control over where the window is positioned. The window contains a scrollable list showing a thumbnail of each annotation within the document (Figure 9). Each

annotation is scaled to between 25% and 100% of the original size – this is to try and fit as much of the annotation as possible in the thumbnail without making it unrecognisable.

The user can switch between position and timeline views of the annotations. This is achieved by changing the sort order of the annotations within the list. The position view uses the line number as the sort and the timeline view uses the time the annotation was first added.

Finally the navigation view can be used to navigate to the actual annotation by clicking on an annotation in the window. This works in a similar way to the collapsed region icon. It includes the automatic scrolling and region expansion and the annotation "flash".

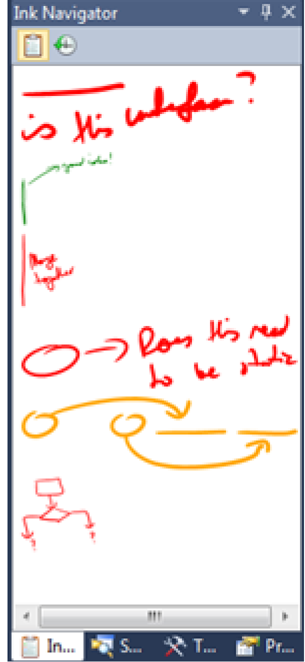

Figure 9: The ink navigation tool window.

5 Evaluation

To assess the usability of vsInk a task-based usability study was carried out. Subjects were asked to perform two code review tasks in Visual Studio 2010 and to annotate 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 a computer science lecturer. All had some experience with Visual Studio with most participants saying they use it frequently. 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. Two rounds of testing were performed – after the first round of testing the major flaws identified were fixed and then the second round of testing was 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 then allowed three minutes to familiarize themselves with vsInk. For 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 they thought the code did not conform to the guidelines. As the task was to evaluate the annotation experience the participant was only given eight minutes to review the code (although they were allowed to finish earlier if desired). After the review was finished an automatic process updated the code and the participant was asked to re-review the code to see if all the issues had been fixed. The researcher observed the participant and noted down any usability issues. In addition a questionnaire was filled in after each review task. After the tasks the researcher and participant went through all the annotations and identified whether the annotation had been correctly re-positioned after the update. Finally there was 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 a number of issues were identified. Before the second round of testing changes were made in an attempt to fix these issues. The main issue found was strokes were being incorrectly added to existing annotations. During the tests the opposite (strokes not being added to annotations correctly) occurred rarely. Therefore the three changes mentioned (see 4.2 above) were made to the grouping process.

The other refinements to vsInk were as follows. Some of the participants complained that the lines were two small or the ink too fat. To fix this the ink thickness was reduced and the line size increased slightly. Another common complaint was the adornments obscured the code. This was fixed by making all adornments semi-transparent and removing non-necessary ones (e.g. the name of the annotator). Participants also mentioned the ink navigator distorted the annotations too much so the amount of distortion was limited to between 20% and 100% of the original size. Observations suggested that the navigation features were not obvious. When a participant selected an annotation in the navigator they did not know which annotation it matched on the document (especially when there were several similar annotations). To fix this the flash was added to identify the selected annotation.

In addition to the issues mentioned above, there were other issues noted that were not fixed due to time constraints. These included: tall annotations disappearing when the anchor point was out of the viewport, cut/paste not including the annotations, and annotations not being included in the undo history.

After the modifications the second set of participants tested vsInk. We found most of the modifications had the desired effect and vsInk was easier to use. However there were still issues with the grouping of strokes into annotations. Using time to group strokes sometimes caused

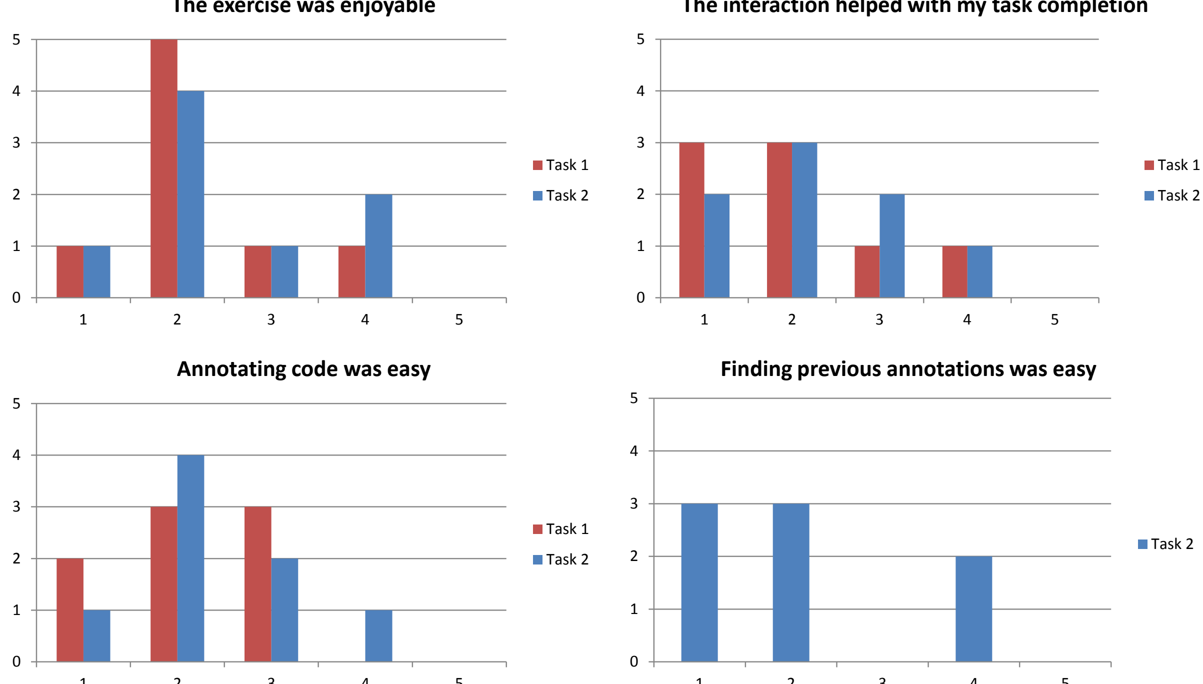

Figure 10: Results from the questionnaires

strokes to be added incorrectly, especially when the participant moved quickly down the code file. The boundary region still caused problems when trying to start a new annotation when the code lines of interest were 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 10). 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).

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

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 and Future Work

With vsInk we have integrated annotation ink into the Visual Studio code editor. The annotations sit over the code and all the existing functionality of the editor is still retained. This is possible because Visual Studio now exposes extension points to allow modifying the actual code editor. However this is still a non-trivial task due to the way the editor works.

The other main technical challenge for vsInk was how to correctly position the annotation relative to the code. Initially the challenge was to integrate with the Visual Studio editor so the annotation always behaved as expected. The first attempts relied on the functionality in Visual Studio doing most of the work. This proved fruitless and we changed to having vsInk do most of the work with the positioning. Once the positioning was working correctly the next challenges were with usability.

The usability study identified two major challenges with combining annotations and code. The first challenge is how to group strokes together into annotations, which if incorrect, in turn causes problems with the re-positioning. The second challenge was tall annotations would disappear un-expectedly.

Previous research has shown that grouping strokes is not easy (e.g. Shilman et al., 2003, Shilman and Viola, 2004, Wang 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). Trying to find a simple set of rules that can handle this variety is always going to result in failures.

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. 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. However 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. 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 usability issue that was not resolved 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. Since $Line_{\#}$ is not visible vsInk hid 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. One approach 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 a flaw with the approach. Since the underlying code can still be edited it was possible to add or delete lines within a tall annotation which broke the anchoring. A decision was made to focus on the grouping instead and this functionality was removed.

One of the interesting suggestions during the user study was to include some way of zooming the interface during inking. This approach has been attempted before with DIZI (Agrawala and Shilman, 2005). 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.

Some possibilities for future work are improving how strokes are grouped together into annotations, handling tall annotations, adding zooming, and implementing missing “common” functionality. Missing “common” functionality includes cut/paste and the undo history. Some fixes for the grouping issue include using contextual information in the rules, using data mining for generating the grouper and using a single-pass grouper. Possible solutions for handling tall annotations include segmenting annotations and adding a mechanism for handling line insertion and deletions. There is also a need for additional user studies around how people would actually use annotations in a code editor.

7 Conclusions

This paper presents vsInk, a tool for annotating program code within Visual Studio using digital ink. This is the first tool that fully integrates digital ink into a code editor in an IDE. It is also the first tool to provide support for annotating within collapsible code regions. The usability study showed that overall vsInk is easy and enjoyable to use. There are two significant issues uncovered during the user study that are yet to be addressed – how to group strokes into annotations and tall annotations disappearing unexpectedly.

Because of the functional deficiencies in earlier prototypes there has been little work on assessing the value of annotating program code. We look forward to exploring real-world user experiences with vsInk.

8 References

Agrawala, M. and Shilman, M. (2005): DIZI: A Digital Ink Zooming Interface for Document Annotation

Human-Computer Interaction - INTERACT 2005. *In:* COSTABILE, M. & PATERNÒ, F. (eds.). Springer Berlin / Heidelberg.

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

Brush, A. J. B., Bargeron, D., Gupta, A. and Cadiz, J. J. (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.

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

Chang, S. H.-H., Chen, X., Priest, R. and Plimmer, B. (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.

Chatti, M. A., Sodhi, T., Specht, M., Klamma, R. and Klemke, R. (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.

Chen, X. and Plimmer, B. (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.

Golovchinsky, G. and Denoue, L. (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.

Lichtschlag, L. and Borchers, J. (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.

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.

Microsoft. *Inside the Editor.* http://msdn.microsoft.com/en-us/library/dd885240.aspx. Accessed 1-Aug-2012 2012.

Morris, M. R., Brush, A. J. B. and Meyers, B. R. (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.

O'hara, K. and Sellen, A. (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.

Plimmer, B., Chang, S. H.-H., Doshi, M., Laycock, L. and Seneviratne, N. (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.

Priest, R. and Plimmer, B. (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.

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

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

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

Tashman, C. S. and Edwards, W. K. (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.

Wang, X., Shilman, M. and Raghupathy, S. (2006): Parsing Ink Annotations on Heterogeneous Documents. *Sketch Based Interfaces and Modeling - SBIM.* Eurographics.

Wilcox, L. D., Schilit, B. N. and Sawhney, N. (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.