ISSUES OF EXTENDING THE USER INTERFACE OF INTEGRATED DEVELOPMENT ENVIRONMENTS

Samuel Hsiao-Heng Chang, Xiaofan Chen, Richard Anthony Priest, Beryl Plimmer
Auckland University
Auckland, New Zealand
{hcha155, xche044, rpri032, beryl}@ec.auckland.ac.nz

ABSTRACT

The current level of extensibility of integrated development environments (IDEs) does not provide sufficient access to make modifications to their user interface components. This limits innovation in IDEs. This paper reviews the problems we have encountered and presents alternative ways to help developers achieve their goals of extending user interface components. Developers interested in extending existing applications will appreciate the information on likely problems and solutions with extensible architectures. Furthermore general suggestions for software architecture extensions to maximize extensibility are included.

Author Keywords

Annotation, integrated development environment, extensibility, user interface

ACM Classification Keywords

D.2.6. Software Engineering, Programming Environments: Integrated environments; D.2.7. Software Engineering, Distribution, Maintenance, and Enhancement: Extensibility; H.5.2. Information Interfaces and Presentation (e.g., HCI), User Interfaces: Graphical user interfaces (GUI)

INTRODUCTION

Modifiability is considered to be one of the most important quality attributes in the study of software architecture[3]. This attribute has been recognized as a basic requirement for many years, one of the specific goals of object oriented programming (OOP) is to promote reuse through encapsulation and inheritance. This architecture is fundamental and is widely used. Many modern software applications are designed to be extendable with a clear Application Programming Interface and plug-in architecture.

The highly customisable nature of modern software allows software developers to focus on the core functions so that the kernel can be kept small and robust. While others can extend the software by making plug-ins to enhance its functionality [2]. Most IDEs follow this trend. IDEs exist to augment a human programmer's ability to create an artifact[24]. Just as individuals have different requirements for a workplace to be convenient for use, there are different requirements for an IDE to ensure the productivity of a software developer[7]. Popular IDEs such as Visual Studio[15], Eclipse[9], and JBuilder[5] all allow third party plug-ins to be installed so the core elements can be extended to support additional functionality and programming languages.

Extensibility also brings innovation. For example, the JUnit testing tool[13] for Eclipse was originally released as a plug-in that integrates into Eclipse. It is now packaged as part of the official distribution, because people accept it as a useful and essential tool in their Eclipse IDE. Other examples of innovations that started as plug-ins include: smooth scrolling[12], session saving [20] and RSS/Atom subscribing [19] for Mozilla Firefox [18].

Plug-in based systems seem to promise almost unlimited customization, however in reality the extensibility is mostly dependent on the extension points provided by the core application [8]. As most IDEs are commercial products, instead of improving the capabilities which are demanded by individual developers, IDE providers often are driven by market and competition [4]. The limitations of plug-ins lie where the software provider has placed or has not placed extension points.

Paradoxically, even though most of the extensions are activated through user interface elements such as buttons, menus and lists, the extensibility of the user interface itself is one of the most often disregarded parts in an IDE in terms of extension points. These user interface elements that can be added are used as a trigger to access the created plug-ins, but rarely to actually alter the user interface. To improve the user interface developers have to create new design windows from scratch to fulfill their needs. An example of this is the Eclipse Visual Editor [1]. There are very few plug-ins which modify the built-in editor itself.

We [23] alluded to some issues of extending Microsoft Visual Studio 2005 in RCA (Rich Code Annotator). RCA aimed to provide an ink annotating tool on top of the IDE. We later found similar issues when we tried to extend

Eclipse SDK 3.3.1 in CodeAnnotator [6]. The limitations of these IDEs prevented us from implementing ink annotation functionality in the intended manner. As a result, an alternative approach was developed, which was not as intuitive to use as we wanted. Thus architecture limitations decrease the usability and cause limitation on innovation.

In this paper, we extend our previous work, specifically exploring the limitation of the user interface of two successful IDEs, Visual Studio and Eclipse. We will compare the problems that they have in common and provide suggested workarounds.

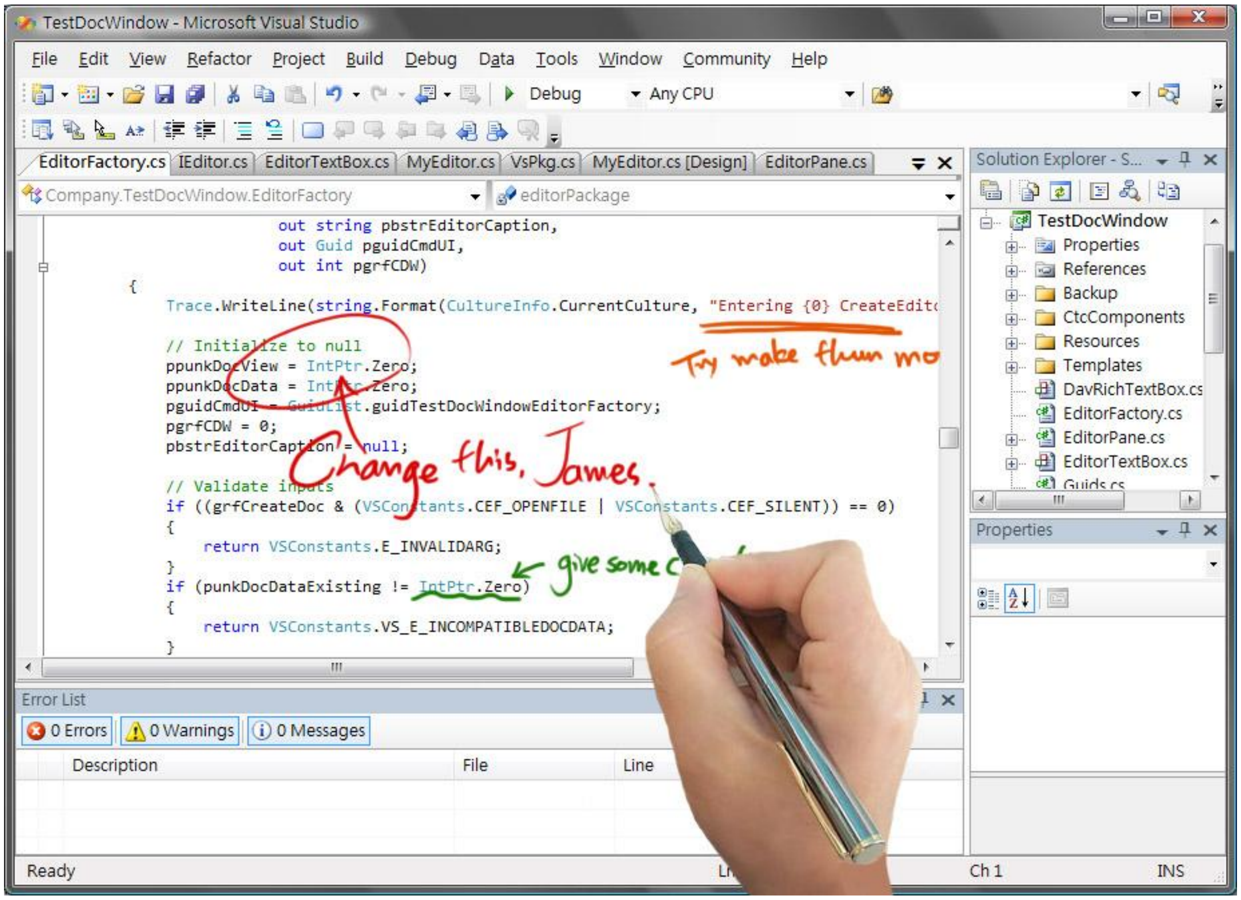
RELATED WORK

Parnas [21] overviewed the issues surrounding extensibility of software in 1979. In this paper, he presented a selection of approaches to extensibility. One of the most important ideas he presented is that flexibility should be included as a requirement when designing the artifact.

Fayad and Cline's research[11] further defined extensibility and the positive effects it will bring to software engineering. These ideas were then discussed by Weck [25] where he used the term "independently extensible" to define a software system to which any individual can develop extensions without the need of knowing the work of other people extending the same artifact. Furthermore the importance of having rules for extensibility is discussed since the extensions should not interfere with each other. Bako et al [2] extended these ideas and discussed and defined the important properties in plug-in based systems:

- Plug-ins can be added at any time (also by end-users).
- Plug-in-based systems offer certain plug-in interfaces (or extension points).
- Plug-ins have a plug-in type, determined through the provided plug-in interface (or extension point).
- Plug-ins are components that can only be used with the application (or environment) they have been developed for.
- Plug-in-based applications can be executed with no plug-ins—having minimal functionality in this case.

The main objective of integrated development environments (IDEs), as stated by des Rivieres and Wiegand [7] is to make programmers more productive. This is achieved by providing tools which are intuitive to use and cover more of the software engineering cycle. In their brief history of IDEs they explain the requirement for IDEs to be extensible so that they can adapt to the ever changing nature of software development. The evolution of IDEs, some of the features and the issues associated with them are discussed by Boekhoudt [4] . He concluded that IDEs



```
                out string pbstrEditorCaption,
                out Guid pguidCmdUI,
                out int pgrfCDW)
    {
        Trace.WriteLine(string.Format(CultureInfo.CurrentCulture, "Entering {0} CreateEdit

        // Initialize to null
        ppunkDocView = IntPtr.Zero;
        ppunkDocData = IntPtr.Zero;
        pguidCmdUI = GuidList.guidTestDocWindowEditorFactory;
        pgrfCDW = 0;
        pbstrEditorCaption = null;

        // Validate inputs
        if ((grfCreateDoc & (VSConstants.CEF_OPENFILE | VSConstants.CEF_SILENT)) == 0)
        {
            return VSConstants.E_INVALIDARG;
        }
        if (punkDocDataExisting != IntPtr.Zero)
        {
            return VSConstants.VS_E_INCOMPATIBLEDOCDATA;
        }
```

Figure 1. Expected implementation of annotation tool in Visual Studio 2005

evolve quickly, not to satisfy developers but to counter competition between rival IDEs. Dietrich et al [8] specifically reviewed Eclipse and offered some suggestions to improve its current extension environment. They suggested a contract language which can be used to specify the interaction between plug-ins. This contract may be used for validating a complex system.

Our current research draws heavily on our previous work on IDE ink annotation tools including RCA, a Visual Studio extension by Priest & Plimmer[23] and CodeAnnotator, the Eclipse version by Chen & Plimmer[6].

MOTIVATION

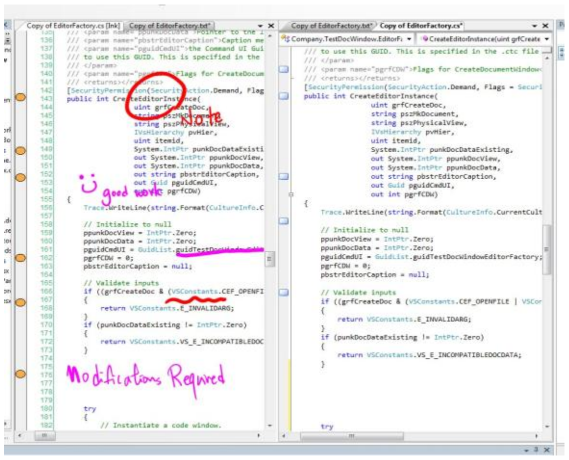
Pen-based computing has brought ink annotation, a traditional human recording method, into the digital world. Research shows that having ink annotating functionality in software is enjoyable [16, 17, 22]. But there is little work done in the area of providing ink annotation functionality in IDEs.

In the past, code review was usually done by having developers examine and annotate on printed copies of the material. As the structure of programs has changed (from linear to OOP) and size and average complexity of projects has grown, the traditional way of reviewing code is no longer feasible. It now makes good sense to have annotating ability in IDEs. Most IDEs provide code handling abilities such as search, replace, outline and so on. The two major goals of an IDE are to increase the productivity of programmer and to cover more of the software engineering cycle[7]. We hypothesize that if the inking capability is successfully implemented, the productivity of developers can be improved through having more distinguishable notes, comments and explanations. Activities such as code review with ink annotation can further extend the use of IDEs in the software lifecycle. Hence both goals of IDEs may be extended by such implementation. However, whether the successful experience of annotation in general can be applied to IDEs has yet to be seen because of the essential difference between an ordinary text editor and an IDE. Our early experiments showed users accept such tools in IDEs[23], but this implementation did not cover all the requirements for the project.

Our goal is to extend an IDE with a plug-in to support ink annotation. By extending the IDE rather than building a separate annotation tool (such as [22]) we can leverage the extensive services of the IDE. We can focus on the implementation of the plug-in and evaluate its efficacy. For this kind of plug-in to be practical and usable, several requirements need to be fulfilled: Technically the plug-in needs to enable annotation on code, annotated data needs to be retained for later use, and the annotated ink must reflow (stay in context with the appropriate data in the base document) when the line is changed or window is scrolled. From a usability point of view, the annotation window should be the same window as the code window (i.e where the code is edited) and the shift between coding and annotating should be smooth and intuitive. This paper represents our third attempt at achieving this functionality.

Our first attempt [23] tried to implement the annotation ability through Visual Studio 2005[15]. In the final solution, when users want to annotate on top of the code, they press a button to create a new tool-window. This copied all the code in the original window as background, and users could annotate on top of the tool-window. Now, users have two windows, one allows annotating where the code can be seen but not modified and another with the code where they could edit the code but could neither annotate nor see the ink.

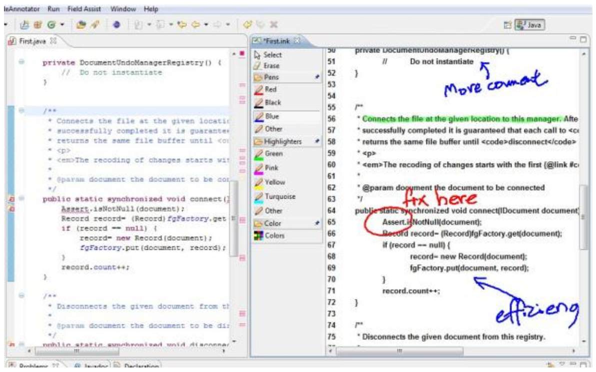
Hence when a user is using the plug-in for code reviewing or commenting, they need to look at the annotation window, find the line number of the commented code, and change to the code window to modify the code.


Copy of EditorFactory.cs [Ink]
Copy of EditorFactory.txt*
Copy of EditorFactory.txt*
Copy of EditorFactory.cs*
Note
good work
Modifications Required

```
/// <param name="pbstrEditorCaption">Caption me
/// <param name="pguidCmdUI">the Command UI Gui
/// to use this GUID. This is specified in the
/// </param>
/// <param name="pgrfCDW">Flags for CreateDocum
/// <returns></returns>
[SecurityPermission(SecurityAction.Demand, Flag
public int CreateEditorInstance(
                uint grfCreateDoc,
                string pszMkDocument,
                string pszPhysicalView,
                IVsHierarchy pvHier,
                uint itemid,
                System.IntPtr punkDocDataExisti
                out System.IntPtr ppunkDocView,
                out System.IntPtr ppunkDocData,
                out string pbstrEditorCaption,
                out Guid pguidCmdUI,
                out int pgrfCDW)
{
    Trace.WriteLine(string.Format(CultureInfo.C

    // Initialize to null
    ppunkDocView = IntPtr.Zero;
    ppunkDocData = IntPtr.Zero;
    pguidCmdUI = GuidList.guidTestDocWindowEditorFactory;
    pgrfCDW = 0;
    pbstrEditorCaption = null;

    // Validate inputs
    if ((grfCreateDoc & (VSConstants.CEF_OPENFI
    {
        return VSConstants.E_INVALIDARG;
    }
    if (punkDocDataExisting != IntPtr.Zero)
    {
        return VSConstants.VS_E_INCOMPATIBLEDOC
    }




    try
    {
        // Instantiate a code window.
```

Figure 2. Implementation by Priest

Our second attempt, on Eclipse SDK 3.3.1[10], ended up with a similar solution. This implementation included pressing a button to create a new tool-window where code is transferred into the background and user can annotate on top of it.



```
private DocumentUndoManagerRegistry() {
        //      Do not instantiate
}


/**
 * Connects the file at the given location to this manager. Afte
 * successfully completed it is guaranteed that each call to <c
 * returns the same file buffer until <code>disconnect</code>
 * <p>
 * <em>The recoding of changes starts with the first {@link #c
 *
 * @param document the document to be connected
 */
public static synchronized void connect(IDocument document)
        Assert.isNotNull(document);
        Record record= (Record)fgFactory.get(document);
        if (record == null) {
                record= new Record(document);
                fgFactory.put(document, record);
        }
        record.count++;
}

/**
 * Disconnects the given document from this registry.
```

Figure 3. Implementation by Chen

Both of the discussed approaches violated one of our requirements: to allow users to annotate with ink strokes in the editing window. It is unintuitive to ask users to shift between two different windows to annotate code and to edit code. Also, as the ink strokes could not be seen in the code window, it is confusing as people need to find line numbers to make modifications accordingly.

VARIOUS APPROACHES

Following our dissatisfaction with the limitations we experienced developing RCA and CodeAnnotator, we have systematically explored extensibility in the two IDEs. In this section the attempts on Visual Studio 2005 and Eclipse SDK 3.3.1 are discussed, including the reasons behind the actions and decisions, and the technical aspects and results.

Visual Studio

Microsoft Visual Studio™ is one of the most widely used IDEs for development in Visual Basic, C++, C# and ASP. It supports extensions through the use of the Visual Studio Software Development Kit (VS SDK). The VS SDK allows the extension developer to extend Visual Studio at three different levels by creating a macro, an add-in or a package.

Macros can only be written in Visual Basic .NET. Since Macros do not have access to the inner workings of Visual Studio, it is considered to be the least powerful way to write an extension. Add-ins are more powerful than macros because they have access through the automation system to the tool windows and command system of Visual Studio. The third option, using packages, is the most powerful extension type VS SDK provides. It allows access to much of the core functionality of Visual Studio. Macros do not provide enough capability to accomplish our goals therefore we will not refer to them further.

Transparent layer over the editor

Our first attempt was with an add-in. The goal was to create a transparent form element which supports annotation, and place it on top of the code window. With the successful implementation of this approach, the annotation could be made on a transparent layer on top of the code, and as the layer would be transparent, the code underneath could be seen by the user. Also, because the transparent layer would be on top of the built-in editor, all the services provided by the built-in editor could still be used. Hence it would allow easy code review, searching and editing. However this approach failed because an extension point could not be found to place a transparent form element (or any other type of form element) on top of the code window.

While exploring the add-in, another problem was encountered; it was discovered that the scrolling of the scrollbar does not fire any event. This problem was considered to be fatal, because even if the form element was successfully placed on top of the code window, it would still be a separate element. Without the scrolling event, when user scrolls the scrollbar, the annotated strokes on top would not be able to be synchronized with the code window underneath.

Customized editor

Because of the inability to fire the scrolling event, we decided to try to create a customized code window. This attempt was via a Visual Studio package. The idea was to customize an editor which has all the required functions including annotation and code editing in the same window, the reflow of ink and the support of language services (syntax highlighting, layout, etc). The built-in editor could then be substituted with this customized editor to provide the desired functions.

An implementation of the customized window has been created with two transparent rich text boxes, one above the other. The top one is for annotating, and the bottom one is for coding. Buttons are provided for user to change layers between editing the code or annotating on top of it. Depending on the choice, the focus of input would be on the appropriate rich text box. Because both windows are transparent, one can always be seen when modifying the other.

By preserving the built-in editor and using a customized editor as an annotation window, this approach can be seen as an extended version of the RCA solution[23], because both annotating and code editing could be done on the annotation window. Also, because the rich text box provided by C# allows changing the font format, the annotation window will have code with the correct syntax coloring if the synchronization between two editors is appropriate. However, there would still be two editors opened at the same time. More critically, we found that the language services could not be used.

By modifying the Global Unique Identifier (GUID) of several components, the customized editor can be opened as the default editor of any selected languages. However if it is used in this way, the user has nowhere to copy the font styles (color and italics) from, thus the style of the text will be the default style unless sets of rules are provided for the rich text box.

The main disadvantage of this approach is that the language services provided by Visual Studio, including syntax coloring, statement completion, brace matching, parameter information tooltips and error markers, are all implemented within the original editor and we have not found a way to abstract them from the built-in editor. Thus there is no way to reuse them. By customizing the new editor, all of the language services need to be rebuilt from scratch if they are to be available. As Visual Studio is not open source software, we cannot copy the language services and use them. We have concluded that the re-creation of these functions was not sensible.

Modifying the built-in editor

To solve the problem encountered with the re-use of language services, the third approach was explored. This involved extending the built-in language specific editors themselves. If we could successfully take the built-in editor out, make extensions on it and substitute the built-in editor with our extended version, essentially we will have achieved the same result as the previous attempt. The difference would be that the language services and other services would be present. However, the only editor related information that could be found was the GUID of editors, which could not be extended because they are not part of the object architecture of C#. This approach failed.

Eclipse

Eclipse[10] is probably as well known as Visual Studio in the world of IDEs. It has an advantage in that it is open source software. Not only does it support extensions by allowing plug-in developments, developers can also modify the core code or examine it to solve problems encountered when extending the platform. To support plug-in developments, the "Eclipse for RCP/Plug-in Developers" or the "Eclipse Classic" versions, both of which are publically available, need to be used.

Build a transparent annotation window

Similar to the implementation in Visual Studio, we made an attempt to build a transparent annotation window that overlays the code window, and has the two windows scroll together. It would have the same advantage as the editor with transparent layer explained in the previous section. Our first attempts at achieving transparency were frustrating: we could either make nothing transparent or make the whole Eclipse window and everything inside transparent.

Due to our inability to get Eclipse to render the transparent layer properly, we could not construct a window where users can review, edit and annotate the code in the same space. As with Visual Studio, we could put the code into the background of a window and ink over it. However, the user still had to go back to the code window to edit the code. In Eclipse, when one window is activated, the other open windows are hidden behind the activated window, so unless the code window is also transparent, it is not possible to view the two windows overlaying each other.

Furthermore, there are complexities in moving the scrollbar of the two windows simultaneously, because the previously mentioned scrolling synchronization problem was also encountered in Eclipse.

Customize an editable code window

The second Eclipse approach we attempted was to create an annotation window that totally resembled the code window and was editable. The expectation with this approach was that the annotation window contained a copy of the code, having the same layout and appearance as the actual code, and users could change code directly on the annotation window with these changes being automatically reflected in the code window. This would keep the code window consistent with the annotation window.

It is easy to copy the code into the annotation window. However, it is difficult to maintain the code's layout and appearance, such as the syntax coloring, because in the built-in editor, each file is displayed by implementing specific layout and appearance strategies. For example, a Java file has its own layout, font and color rules to display the code in Eclipse. Unlike the rich text box in Visual Studio which will copy all style information of fonts, under Eclipse, these rules cannot be preserved when copying the code to other windows. Also, because of the lack of reusability, these rules would need to be re-implemented to properly display the copied code in the annotation window.

Furthermore, if we extend the annotation window from the code window, the annotation window could only deal with text, not figures or drawings. This is because the code window is actually a text editor in Eclipse. If we extend the annotation window from Eclipse Graphical Editing Framework (GEF), it is complicated to program rules since GEF deals with everything as a figure, but not as text. The result of this is that the language services would have to be ignored.

Source recompile

We considered, and briefly explored, modification of the core source code and recompilation of Eclipse, because essentially this should allow maximum modifiability to the IDE. However this is a complex process and it nullifies the extensibility of the architecture model.

Points of failure

From all the work reviewed above, we found that some common problems exist within the extensibility of the two selected IDEs.

The first problem is that no extension point can be found on the editor window to add a new element. The editor window can be reached to get information such as the text within or a line number; however the interface of it can hardly be altered.

The second problem is that although customizing an editor is possible, the language services are all compactly built-in, which causes difficulty in reusing them. It is a waste of work to rebuild all the features when wanting to extend the editor of a certain language.

There are also problems associated with the event firing of scrollbars. Scrolling does not trigger any event, which means that synchronization of the code and annotation layers is challenging.

Extension possibilities

The limitations we uncovered not only restrict the development of our annotation plug-in, but can also limit innovations in many applications which are used in areas other than ink annotation. Moreover it may be useful to integrate diagrams and code, effectively blending visual programming techniques with text-based programming. Mechanisms for placing drawing elements in a code window are required to implement this. Our experience suggests that this would result in the same problems as we encountered with digital ink annotation.

Users of the IDEs may also want to redefine or extend the filter of syntax coloring. For example, under current support, comments can only have one fixed color and font, but it would certainly be helpful if user can set special colors and fonts for some important comments to make them stand out from other comments. Current extensibility would require a rebuild of the entire language service to achieve this.

The ability to fire a scroll event is particularly useful. This problem was also encountered in the work of Plimmer & Mason when implementing a pen-based annotation application [22]. Scrolling events are needed to ensure the synchronization of the two windows or anything associated with information based on the position of the code window.

ALTERNATIVE APPROACHS

We have explored two alternative approaches. Both of these are workarounds and do not provide an ideal user experience.

Separate windows

The common solution presented in previous work [6, 23] is to create a new window which solely focuses on annotating. This approach solves several problems.

Both Visual Studio and Eclipse provide functionality for developers to make windows for use of tools such as the visual editor. The windows are referred as tool windows, and can be treated as forms. Thus the annotation window can be built on a tool window using form elements, which gives flexibility that the original editors do not allow. A rich text box can be created to contain the code copied from the code window, and on top of it, another layer can be created for annotation.

The problem of editor scrolling is also solved since the element that is used is completely flexible. Either the scroll bar form element can be used with rules to define how it should react with the code and annotation window, or the rich text box can be extended so that it fires an event when scrolled.

However, new problems arise in this solution. When people edit the code window, the modifications they have made are expected to be visible when they switch back to the annotation window, with the ink reflowed according to their relative position to word or paragraphs, and even the position of scroll bars, and vice versa if the annotation window allows modification in code. It is intuitive to have this synchronization at the time of switching and window closing.

RCA and CodeAnnotator both blocked the ability for users to edit code in the annotation window (although this is possible), because it is then difficult to keep consistency between the annotation window and the code window. The reason is that the language service mechanisms are not provided by the annotation window. If code editing was supported in the annotation window, syntax coloring would be lost: new text would have to use the same style as the previous character regardless of its purpose because font and colors are not based on the language services. This problem could be solved by either building language services on the annotation window, or by providing more frequent synchronization of the two windows so that when a user edits in the annotation window, the modifications are copied to the code window and then the result is returned with updated style information to the annotation window.

From a usability point of view, this solution is not as intuitive as to have an annotation layer directly on top of the built-in editor. However under current technology, this is certainly the simplest and most applicable way to solve problems in this category.

Customized editor

Another alternative is to replace the built-in editor with a customized editor. Both Visual Studio and Eclipse IDE support the customization of new editors, which is proven by the extensibility of new languages such as Python on Visual Studio and Ruby on Eclipse. The customization of a new editor can be done in the same way as building form elements. It gives high flexibility like the solution above, but prevents the redundancy of switching between windows as the annotation window can be easily built into the editing window.

However, with the current extensibility of both IDEs, there is no easy way of reusing the language services. The only solution is to re-implement them all from scratch, which is a task that could be avoided with better design of the software architecture.

This solution may be the best for the usability of the software; however it is troublesome and demands a lot of work from the developers.

DISCUSSION

We hoped to implement a plug-in for an extensible IDE that allows the user to annotate on top of a code editor similarly to the ink function in Microsoft Word [14]. Visual Studio and Eclipse were the targeted IDEs because of their reputation and extensibility, but obstacles were encountered during the implementation phases. Different approaches were taken but numerous problems were encountered because of the incomplete support of user interface modifiability. As mentioned in previous sections, the lack of extension points in the user interface component and the low reusability made the completion of this task challenging. This may be due to the fact that no one has attempted to extend the IDEs in such areas in the past. The user interfaces appear to be designed without the flexibility for customization. It may be possible to write an add-in that supports the types of extensions that we would like to achieve. However, we could not identify an elegant solution.

After many failures we decided to solve the problem in ways which are comparatively clumsy and unintuitive. These alternative solutions have been recorded in previous sections as they may provide information for people who intend to build extensions in similar areas. The final implementations of RCA [23] and CodeAnnotator [6] resulted in similar solutions: to separate the coding and the annotation window, and synchronize between them. This result may have arisen because both of the selected IDEs have similar sets of limitations. In further research, we discovered that it is hard to modify the built-in editors in both of the IDEs, but it is significantly easier to make a customized editor or window.

The limitations we uncovered may be resulted from one of the requirement of extensibility of an IDE: developer should be able to integrate support of a new language into the IDE. As when integrate a new language, most of the language services may need to be re-created, it is logical to have developers define their own set of language service syntaxes instead of using the original ones. Also, the high flexibility of customizing the tool windows allows the creation of many different extensions since there are virtually no restrictions on it. However those supports are far from enough, if a developer would wish to extend the interface design of an editor on a supported language.

In the past, changing the user interface design of a built-in editor may have been considered as unnecessary since traditionally, as long as the user could quickly find the sentence or word needed and the supported language service such as syntax highlighting or text completion is provided, the editor would be considered as good. Editors were used for editing text, and there was no obvious reason for people to be motivated to place things on top of them.

The definition of a good editor may have altered in recent years, just as the definitions of a good word processor or a browser have changed. The user interface is experiencing rapid changes brought about by innovations in hardware and software. As such it should be considered an important area where customization in existing software is enabled. User evaluation studies of RCA and CodeAnnotator [6, 23] have already drawn positive feedback from users who tried using the plug-in for annotating on IDEs, even though they are not optimal. There may be more creative ideas that are not realized because of the lack of extensibility.

Open source software seems to promise a lot, with the ability to let developers look into the source code and make modifications. However, most of the time they are not easier to extend than closed source software because of the tangling between classes and the lack of proper documentation. If innovations cannot be built as plug-ins, they are harder to disseminate and less people will use them. This, in turn, creates difficulty in getting meaningful feedback.

An extensible architecture should support the ease of modifiability and adaptability of a system. However, most of them are limited. It is understandable that a system cannot support all possible extension opportunities, but the user interface element certainly should be valued higher than it is, because it is the most important part of an IDE, the first thing users see and the thing they spend the most time with.

Extensibility and the user experience should be evolutionary. The architecture of IDEs should be modified to allow more flexibility as requirements arise. As technologies such as touch-screens become commonplace, not only IDE developers, but the developers of all kinds of software can consider different perspectives on the extensibility of their products. The providers of IDEs should consider exposing extension points in more user interface components in future releases.

SUMMARY AND FUTURE WORK

Taken together, the experiments demonstrated that the extensibility of Visual Studio and Eclipse in the area of extending the user interface elements are unsatisfactory.

Major problems include: the lack of an extension point on the editor window, the missing event from scrolling the window and the limitations on reusing language services and built-in editors. These problems prevent innovation. Suggestions of alternative approaches to specific problems are discussed, including the customization of editors, which gives more flexibility, and the making of a tool window which can synchronize with the built-in editor, hence imitating existing language services. We conclude that the extensibility should be improved, especially as new hardware provides more natural human interaction possibilities.

Further research can be done in several areas to allow the realization of our primary goal: ink annotation on the code window. The modification of the core Eclipse IDE code is one possibility. To do this would require detailed knowledge on its underlying structure. Alternatively, Visual Studio 2008 (we used Visual Studio 2005) may have improvements to the extensibility. However, from cursory examination this does not look promising.

REFERENCE

1. *Visual Editor Project* [cited 2008 Feb 21]; Available from: http://www.eclipse.org/vep/WebContent/main.php.
2. Bako, B., et al., *Plugin-Based Systems with Self-Organized Hierarchical Presentation.* Software Engineering Research and Practice, 2006: p. 577-584.
3. Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice*. 2003: Addison Wesley Professional.
4. Boekhoudt, C., *The Big Bang Theory of IDEs*. Queue, 2003. **1**(7): p. 74 - 82.
5. Borland. *JBuilder product page*. 2008 [cited Feb 21]; Available from: http://www.codegear.com/products/jbuilder.
6. Chen, X. and B. Plimmer. *CodeAnnotator: Digital Ink Annotation within Eclipse*. in *OzCHI 2007: Entertaining User Interfaces* 2007. Adelaide: ACM.
7. des Rivieres, J. and J. Wiegand, *Eclipse: a platform for integrating development tools*. . IBM SYSTEMS JOURNAL, 2004. **43**(2): p. 371 - 383.
8. Dietrich, J., J. Hosking, and J. Giles, *A Formal Contract Language for Plugin-based Software Engineering*. Engineering Complex Computer Systems, 2007.

9. Eclipse.org. *Eclipse.org home*. 2008 [cited Feb 21]; Available from: http://www.eclipse.org/.
10. Eclipse.org. *Eclipse.org home*. [cited 2008 Feb 21]; Available from: http://www.eclipse.org/.
11. Fayad, M. and M.P. Cline, *Aspects of software adaptability*. Communications of the ACM, 1996. **39**: p. 58 - 59
12. Halachmi, A. *SmoothWheel*. [cited 2008 Feb 21]; Available from: http://smoothwheel.mozdev.org/.
13. IBM. *JDT JUnit integration*. [cited 2008 Feb 21]; Available from: http://publib.boulder.ibm.com/infocenter/radhelp/v6r0m1/index.jsp?topic=/org.eclipse.jdt.doc.isv/guide/jdt_int_junit.htm.
14. Microsoft. *Microsoft Word*. [cited 2008 Feb 21]; Available from: http://office.microsoft.com/en-us/word/default.aspx.
15. Microsoft. *Visual Studio* [cited 2008 Feb 21]; Available from: http://msdn2.microsoft.com/en-us/vstudio/default.aspx.
16. Mock, K., *Teaching with Tablet PCs*. Journal of Computing Sciences in Colleges, 2004. **20**(2): p. 17-27.
17. Moran, T.P., P. Chiu, and W. van Melle. *Pen-Based interaction techniques for organizing material on an electronic whiteboard*. in *10th Annual Symposium on User Interface Software and Technology*. 1997. Banff, Canada: ACM.
18. Mozilla. *Firefox web browser*. [cited 2008 Feb 2]; Available from: http://www.mozilla.com/en-US/firefox/.
19. Mozilla. *The info RSS*. [cited 2008 Feb 2]; Available from: http://inforss.mozdev.org/.
20. Mozilla. *Mozilla Firefox 2 Release Notes*. [cited 2008 Feb 21]; Available from: http://en-us.www.mozilla.com/en-US/firefox/2.0/releasenotes/.
21. Parnas, D.L., *Designing Software for Ease of Extension and Contraction*. Software Engineering, IEEE Transactions on, 1979. **SE-5, Issue: 2**: p. 128- 138.
22. Plimmer, B. and P. Mason. *A Pen-based Paperless Environment for Annotating and Marking Student Assignments*. in *AUIC*. 2006. Hobart: CRPIT.
23. Priest, R. and B. Plimmer. *RCA: Experiences with an IDE Annotation Tool*. in *CHINZ*. 2006. Christchurch: ACM.
24. Robbins, J.E., D.M. Hilbert, and D.F. Redmiles, *Extending Design Environments to Software Architecture Design*. Automated Software Engineering: An International Journal, 1998. **5**: p. 261--290.
25. Weck, W., *Independently Extensible Component Frameworks*. Special Issues in Object-Oriented Programming, M. M¨uhlh¨auser,Ed., Heidelberg, 1997: p. 177–183.