View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ScholarBank@NUS

A WYSIWYG ADD-ON DEVELOPMENT ENVIRONMENT FOR THIRD PARTY SOFTWARE APPLICATIONS

ZHANG ZHONGYUAN

(B.Eng.) Tsinghua University, China

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2012

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Acknowledgements

First of all, I would like to express my deepest gratitude to my supervisor Dr. Shengdong Zhao for his guidance and support to my study, as well as my life. He is not only an excellent HCI researcher, but also an excellent mentor. I was fortunate to be his student.

Additionally, I want to thank all my collaborators in the past one year: Dr. James Eagan, Dr. Ramanathan Subramanian, Melissa Wong, Kristal Chan, Niti Madhugiri and Mengyao Zhao.

Furthermore, thank all members of the NUS-HCI lab. Everyone in the lab is always willing to give me a hand as needed. Together, they make the lab a great place to work.

Finally, I am deeply grateful to my family. They give me their most sincere love, support, and encouragement all the time.

Table of Contents

Declaration i

Acknowledgements ii

Table of Contents iii

Summary vi

List of Tables vii

List of Figures viii

Chapter 1 Introduction 1

1.1 Background 1

1.2 Motivation 3

1.3 Structure of the Thesis 4

Chapter 2 Related Work 5

2.1 Overview of Add-on Architectures in Existing Software 5

2.1.1 Implementation Mechanisms 5

2.1.2 Development Environment 10

2.1.3 Summary 11

2.2 General Add-on Architectures for Third-Party Applications 12

2.2.1 Surface-Level Modifications 12

2.2.2 Program Behavior-Level Modification 16

2.3 Summary 21

Chapter 3 Proposed Approach 23

3.1 Introduction 23

3.2 System Architecture....25

3.3 Runtime Intervention through DLL injection....27

3.4 Modifying GUI properties....30

3.4.1 Retrieving GUI Information....31

3.4.2 Modification and Addition....31

3.4.3 Deletion....31

3.4.4 Modifying program behaviors....32

3.5 Supports for the GUI Editor....32

3.5.1 Inter-Process Communication....33

3.5.2 Creation of Project in IDE....37

3.5.3 Code Conversion....40

Chapter 4 Utility Add-ons....45

4.1 Property Editor....45

4.2 Interaction Logger....46

4.3 Multi-stroke Marking Menu....46

4.4 Heat Map Generator....47

4.5 Summary....48

Chapter 5 User Study....49

5.1 Tools....49

5.2 Participants....51

5.3 Apparatus....51

5.4 Experimental protocol....51

5.5 Quantitative Measures 53

5.6 Qualitative Analysis 54

5.7 Summary 58

Chapter 6 Discussion 60

6.1 Extension to other frameworks and platforms 60

Chapter 7 Conclusion 64

7.1 Contribution 64

7.2 Limitations 65

7.2.1 Custom Widget Clone and Modification 65

7.2.2 Dynamic Widgets 66

7.3 Future Work 66

Bibliography 67

Summary

Software rarely fulfills the demands of all users in its initial development stage. Individual needs, which include both interactive preferences and functional requirements, differ in users and often change over time. Mindful of the importance of making software adaptable to individuals, developers typically could enhance their software by allowing reconfiguring user interfaces and/or add-ons that can modify software behaviors, leaving the original main program unchanged. However, many software applications support limited or no add-on architecture, due to additional overhead in software design, development, and maintenance.

This thesis presents the WADE IDE, which enables easy modification of GUI-based software applications without access to their source code. WADE retrieves the host application's GUI hierarchy by injecting a dynamically-linked library (DLL) into the host program, and converting this information to a declarative language, thereby enabling GUI modifications in a WYSIWYG fashion through a GUI editor. The GUI editor also provides direct association of event handlers with GUI widgets, greatly simplifying the job of modifying not only appearance but also software behavior. We demonstrate the usefulness of WADE through (a) the implementation of add-ons that require deep changes to existing software and are difficult to realize via other approaches and (b) a user-study.

List of Tables

Table 2.1 Dot file command of Vim 7

Table 2.2 Comparison between previous approaches and WADE 22

Table 3.1Cache file solution 34

Table 3.2 Windows Forms project files 40

List of Figures

Figure 2-1 Interface customization panel of Visual Studio 2010 6
Figure 2-2 Themes of WordPress 8
Figure 2-3 ".addin" file of SharpDevelop 11
Figure 2-4 Minimizing all windows in Sikuli 14
Figure 2-5 Bubble Cursor 15
Figure 2-6 Users' recent manipulation histories 16
Figure 2-7 Facades 17
Figure 2-8 Extending the window management using DiamondSpin 19
Figure 3-1 Steps of adding a Batch Image Conversion add-on to Paint.NET. 24
Figure 3-2 Architecture overview of WADE 26
Figure 3-3 Common Language Runtime in .NET framework 29
Figure 3-4 Creating CLR using C++ 29
Figure 3-5 SharpDevelop class hierarchy 39
Figure 3-6 SharpDevelop design view 39
Figure 3-7 Code conversion example one 43
Figure 3-8 Code conversion example two 43
Figure 3-9 Code conversion example three 44
Figure 4-1 PropertyEditor add-on 45
Figure 4-2 EventRecorder add-on 46
Figure 4-3 MarkingMenu add-on 47
Figure 4-4 HeatMapGenerator add-on 48
Figure 5-1 Screenshot of Managed Spy 50
Figure 5-2 Work flow of WADE and Scotty-like approaches 57

Chapter 1 Introduction

1.1 Background

Software rarely fulfills the needs of all users all the time. Software systems are complex, frequently having to satisfy conflicting requirements and constraints. As such, designers optimize their software for a narrower class of users and a narrower subset of the problem. Since individual users' preferences and interactive needs can change over time, it is essential for software tools to be user-adaptable in order to effectively cater to these ever-changing requirements (Mackay 1991, Robinson 1993).

Mindful of the need to make software adaptable to individual needs, developers typically allow for software customization by providing:

- Capabilities for ***reconfiguring*** existing features and functions to suit personal taste such as via preferences panes or dot files; or
- A software architecture for incorporating ***add-ons*** — additional functionalities that enhance/modify the behaviors of the original application using add-ons, *plugins*, *scripts* and/or *extensions*.

To illustrate the demands and necessity of add-ons, we present four usage scenarios where end users will need the power of reconfiguring interfaces and add-ons:

- **Reconfiguration**: Albert's favorite photo editor includes buttons to share his works to the Mybook, Facespace, Doodle+, and Failwhale social networks, but he only uses Failwhale. He is overwhelmed by clutter of the extra features and wants to remove unused icons from the toolbar and enlarge the Failwhale icon to make it easier to acquire. Albert has little programming knowledge

and cannot hack into the code by himself. He searches the Internet and finds no specific add-ons to make such changes, but he does find a general GUI property editor add-on. By installing the add-on to the photo editor, he removes extra buttons and enlarges the Failwhale icon.

- **Language localization**: Kevin can create incredible photo-realistic effects with the image manipulation tools in Paint.NET, and he teaches some tricks to his Russian friend, Ivanov. Ivanov finds some of the tools and effects very cool, and wants to create some visual effects on his own. However, Ivanov is not comfortable with English language commands, and would prefer a Paint.NET GUI with Russian labels instead. Unfortunately, Russian is not among the languages supported by Paint.NET, so he uses an English-Russian translator to create an add-on to synthesize the Paint.NET GUI in Russian. He modifies the text of the relevant toolbar labels sing the GUI editor and shares the add-on on the Internet for the benefit of others.
- **Customization for the elderly**: John wants to modify the interface to his word processor so that it can be easily used by his father, who is over 70 years old and has relatively poor eyesight. John's father uses only a specific set of GUI functions, but would like those widgets to be clearly visible on screen (e.g., at a much larger size). Upon installing a GUI property editor add-on, John is able to easily hide functions unlikely to be used by his father, as well as enlarge the size of the relevant widgets so that they are easily locatable on screen.
- **Creation of software variants for testing novel interaction techniques**: Mary is a user interface researcher. She has heard an unusual number of complaints about the most recently released version of a popular software application. After reviewing the application's design, she identifies three possi-

ble problems and comes up with several possible improvements. However, in order to confirm her hypotheses, she needs to conduct user studies to compare the original interface with her proposed enhancements. She uses an add-on to make her changes to the original interface, and installs a generic interaction logger add-on to collect data from the original and revised interfaces. By performing a series of studies, Mary identifies the exact enhancements that can help improve the software's usability.

1.2 Motivation

Since in many scenarios and cases, add-ons are required by end users, some applications provide their add-on architecture in different ways, e.g. configuration panel, dot file, skin / theme, functional libraries, startup libraries. While all of these approaches can provide users with a great deal of control, (i) there exists a trade-off between an adaptation's expressiveness and user skill/effort required to realize it, and (ii) every approach requires the developer to provide a certain degree of explicit support for customization. Preferences and dot files require the developer to explicitly make multiple variants of some functionality and to provide a configuration interface. Plugins, scripting interfaces and extensions require the developer to provide and maintain an external API to their software, which may potentially require maintaining a separate interface to internal functionality.

Owing to the above issues, many software developers do not provide support for add-ons. Even when they do, such support is often limited. Much research has focused on approaches that enable third-party developers to modify the interface or behavior of existing applications without access to source code or an external API. These approaches typically work by either: 1) operating on the surface-level of the interface, intercepting the pixels output to the screen and input events before they are delivered

to the application (Dixon and Fogarty 2010, Stuerzlinger, et al. 2006); or 2) integrating with the toolkit to gain access to internal program structures (Eagan, Beaudouin-Lafon and Mackay 2011, Edwards, Hudson, et al. 1997). While these methods provide some way for a third-party developer to enhance / modify existing applications, the third-party developer still requires a deep understanding of the relevant parts of the system in order to realize the desired behavior. The deeper an approach peers into the implementation of the host application, the deeper this understanding may need to be. It is, therefore, worthwhile to explore alternative methods which will enable users to modify applications with little understanding and effort.

1.3 Structure of the Thesis

The rest sections of this thesis are structured as follows:

- Chapter 2 briefly introduces add-on architectures of famous applications, and reviews previous efforts of building general add-on architectures.
- Chapter 3 describes the approach proposed in this thesis, called WADE.
- Chapter 4 lists several add-ons which were developed under WADE's architecture, to show the capacities of WADE.
- Chapter 5 presents a user study to demonstrate the efficacy of WADE.
- Chapter 6 discusses how to extend WADE to other frameworks and platforms.
- Chapter 7 concludes this thesis by summarizing its contributions, limitations, and some possible future directions.

Chapter 2 Related Work

2.1 Overview of Add-on Architectures in Existing Software

Some existing applications were designed to allow built-in reconfiguring or add-on architectures. These applications, which are called skins, themes, plugins, extensions, or scripts, provide different degrees of freedom to the users, including changing font, colors, and texts, or adding images, hiding items, relocating widgets, and replacing widgets. Some applications also support add-ons to expand functionalities of software. In this section, we review some famous applications, including web browsers, office suites, text editors, graphics editors, Integrated Development Environment (IDE), and web utilities, that have add-on architectures. Their implementation mechanisms and add-on development environments are summarized next.

2.1.1 Implementation Mechanisms

- **Configuration Panel**. The built-in configuration panel / dialog / menu is one of the earliest approaches providing customization ability. Users can access these predefined options in main menus or right click context menus. This approach is mostly applied for setting visibility or layout of components. For example, Microsoft Visual Studio 2010 provides a customization panel for setting components of menu bar, toolbar, and context menu, as shown in Figure 2.1. Microsoft Office Suite 2007 allows customization of the items shown in the Quick Access Toolbar through a configuration dialog. Additionally, in many web browsers, the user can decide which toolbars will be shown at the top using a context menu.

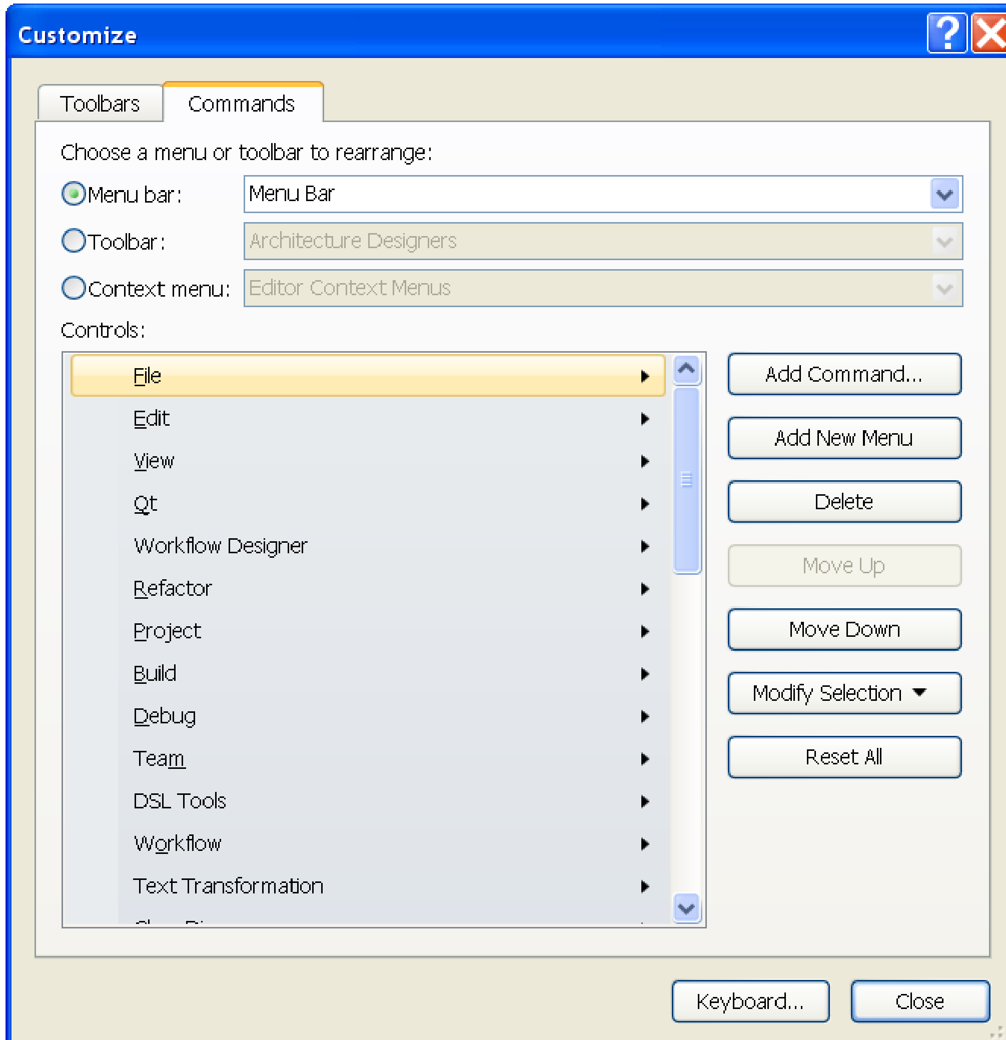


Figure 2-1 Interface customization panel of Visual Studio 2010

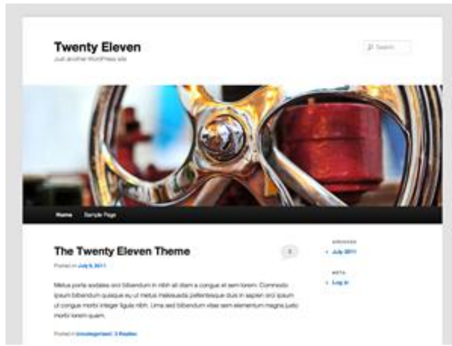
- **Dot File**. Dot file is favored by many applications that originated in Unix-like systems, which are less user-friendly but more flexible than visual configuration panels. Dot files are usually text files whose filenames start with dot symbol, which means hidden files in Unix-like systems. Software can save user settings and data, including UI or functions, in these dot files, which are located in separate or central folders within the user's home folder (Russell, Quinlan and Yeoh 2004). Although not all setting files start with dot in filenames (e.g., many applications that originated in Windows system), they play

the same role. Table 2.1 shows an example of dot file *".vimrc"*, the configuration file for text editor Vim.

Text	**Meaning**
set nu	Show line number before each line
set tabstop=4	Set tab width to 4 space
set expandtab	Expand all tab with equivalent whitespaces
set autoindent	Auto indent new line according to its previous line

Table 2.1 Dot file command of Vim

- **Skin / Theme**. As a complementary solution to a configuration panel, skin / theme templates focus on changing images or textures of existing visual widgets or replacing drawing methods of interface widgets. Skin was initially used in video games (e.g., Quake) to allow players change the appearance of characters (Stuerzlinger, et al. 2006) and was later introduced to media players and some other software. Themes or skins may also be more flexible because they allow a user to change visual styles. For example, in WordPress, an open source blogging tool, there are many themes for users to download and install (Silver 2009), which provides a variety of visual styles (Figure 2.2).



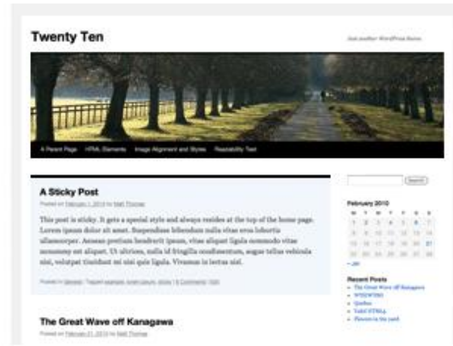


Figure 2-2 Themes of WordPress

- **Functional Library**. All of the previous three approaches can only change the appearance of interfaces. However, in many cases, new functions are necessary for applications. Publishing new versions could be a straightforward solution, however, different functions may be needed by different users. Requiring all users to upgrade to a new version that contains several new features, when most users may only need one or two, is definitely not a smart so-

lution. Moreover, new features may be very trivial, which makes upgrading the entire software not feasible. As a typical example, graphic editing software may continuously encounter new increasingly popular image formats that are supported by the initial version. To deal with this issue, many applications separate some functions into libraries, which could be individually updated while leaving the majority of the software unchanged. These libraries are called by the main program to provide services for the program through explicitly maintained APIs. The libraries could be provided with original software or developed by third party developers to extend the capability of applications.

Returning to the previous example, graphic editing software may check all libraries in a file format folder to find a correct parsing function, when opening an image file. The parsing function converts external image files to an internal image editing format for the program. When saving images, a similar process occurs. As an instance, this approach is adopted by Paint.NET, a free graphic editing software application that runs on the Microsoft .NET framework. Paint.NET checks all libraries under its *"FileTypes"* folder when opening and saving non-default image files. Moreover, Paint.NET also uses this approach to support extensible effect editing functions. Under Paint.NET, there is an *"Effects"* folder, where effect libraries are located. Each time users click on the "Effects" menu item, Paint.NET checks all libraries in the *"Effects"* folder and adds all legal effects to the menu list. If users then select a specific effect, the current edited image is passed to corresponding library function (Dietrich n.d.). This approach largely enhances the flexibility of software.

- **Startup Library**. The most significant difference between a startup library and a functional library is that the latter is loaded at the startup of applications. On the other hand, the former approach (e.g., Paint.NET) only calls libraries when users trigger some events, the disadvantage of which being that add-ons cannot actively change interfaces or behaviors of the program. Instead, the startup library approach calls libraries' initialization functions during the startup period of applications. These libraries can freely change existing components as long as access is permitted. For example, an open source text editor Notepad++ adopted this approach (Wu 2010). *"CommandMenuInit"* method is called at the startup of Notepad++ and has full access to program resources.

2.1.2 Development Environment

To develop the aforementioned add-on architecture solutions, several approaches exist in current software:

- **Built-in Panel**. For the *Configuration Panel* and *Dot File*, all work is done by application providers. Interfaces to configure widgets or functions to parse dot file are all implemented in original applications. Third party developers do not need to and cannot extend the extension ability. However, users or application providers could share their dot files (templates).
- **Declarative (Custom) Language / Format**. For skin / theme templates and library approach, applications often require developers to use certain declarative languages to configure application UI. These languages could be standard ones, e.g. XML (Bray, et al. 1997), CSS (Lie and Bos 1997), or their own custom format. For example, an open source IDE SharpDevelop uses an *".addin"* file to define interfaces. As shown in Figure 2.3, developers can set

the assembly (library) name, menu item text, and handler functions in the *".addin"* files. Handler functions should be compiled into a library and are called when the menu item is clicked.(Holm, Kruger and Spuida 2004, Georgescu and Milodin 2010)

```xml
<AddIn name="PluginHelper" author="hci" url="" description="TODO: Put description here">
  <Runtime>
    <Import assembly="PluginHelper.dll" />
  </Runtime>
  <Path name="/Workspace/Tools">
    <MenuItem id="PH_MenuRoot" label="SWADE" type="Menu">
      <MenuItem id="PH_StartListening" label="Start Listening" class="SDPlugin.ToolCommand_StartServer"/>
      <MenuItem id="PH_GeneratingCodes" label="Generate Library Project" class="SDPlugin.ToolCommand_CodeGenerator"/>
    </MenuItem>
  </Path>
</AddIn>
```

Figure 2-3 ".addin" file of SharpDevelop

- **Coding-based**. Unlike declarative languages, in some add-on architectures, developers directly use a programming language, usually the same language used by the original software, to write modifications and functions of the program. This is mostly seen in the startup library approach, where library initialization function is called at program startup to perform modifications. Program resources are usually packed in some singleton classes that add-ons can access globally in the program. As an example, Notepad++ is adopting this approach. To simplify the work of add-on developers, sometimes add-on project templates are provided by application providers or third party. In the templates, descriptions and examples were given to guide developers to be used when creating specific effects.

2.1.3 Summary

As previously mentioned, different approaches to support add-on architecture have been explored. *Configuration Panel* is the most user-friendly and can be used by end users, but usually supports predefined and limited configurations. *Dot File*, which is more flexible but less user-friendly, allows users to share their configurations easily.

Skin / Theme is convenient to install, however, users have less control. All three approaches can be implemented to directly serve end users, but they only allow settings of UI. The *Functional Library* is a common way to add new functions. Unfortunately, the three approaches for UI setting and using the *Functional Library* all require predefined interfaces provided by application providers, which creates significant overhead of software design and implementation. Moreover, these predefined interfaces rarely fulfill the demands or future extension requirements of all users. The *Startup Library* overcomes this disadvantage, since libraries have full access to program resources, working like a normal initialization method. Meanwhile, the support for *Startup Library* requires little efforts to implement. This approach also has the significant disadvantage that most startup libraries are pure coding-based programming, which means, unlike developing standalone applications, programmers do not have the help of GUI editors when they want to modify UI of applications.

2.2 General Add-on Architectures for Third-Party Applications

Various methods to support third-party application modifications have been pursued. The next section examines representative approaches and divides them into two categories: surface-level modifications and program behavior-level modifications.

2.2.1 Surface-Level Modifications

Surface-level modifications do not rely on any particular support from application providers. Instead, they operate on the interface that is presented to the user and the input events that he or she provides. More specifically, they take pixels rendered on screen and events from the keyboard and mouse as input of the system, while the output usually involves re-rendering the screen.

Virtual Network Computing (VNC) proposed by Richardson et al (Richardson, et al. 1998) is an attempt to teleport pixels from a partial or entire screen from server to

clients. It requires a server, usually a home computer or a work place computer, to run target applications. Users could remotely receive screen pixels from the server and send mouse or keyboard events back to the server, using a thin client and network with server. VNC server takes screen pixels as input so that it does not need any information or support from target applications. Transmission of pixels uses standard network protocols like TCP/IP. And to render screen on clients is not harder than playing a video. More efforts, however, should be made to optimize performance and security. Thus, VNC can work with different interfaces and operating systems without limitations of distance. However, VNC does not allow adjusting of existing interfaces or incorporating of new functions to applications.

Adopting similar techniques, D. S. Tan et al. proposed WinCuts (Tan, Meyers and Czerwinski 2004) to allow users to replicate arbitrary regions of running windows to new independent windows. The goal of WinCuts is to allow better usage of limited screen space to display more interested information simultaneously. Microsoft Visual C++ .NET and Win32 Graphics Device Interface (GDI) API were utilized to build the system, running on a Windows XP system. For remote representing, PNG image compressing and peer-to-peer socket communication were used. This technique, which provides more flexibility and applicable scenarios than VNC, still stand within the scope of representing existing interfaces without improving them.

Several techniques have been proposed to adaptively manage windows (Miah and Alty 2000, Hutchings and Stasko 2002, Kandogan and Schneiderman 1997). Most modern graphic-based operating systems provide their windowing systems with many features to users. Users can open multiple windows to concurrently work on several tasks. These windows can even connect to a remote machine. The core responsibility of a windowing system is to manage these windows efficiently. If windows are not managed well, the desktop may be cluttered with windows, making it difficult for

users to easily locate or open target windows. Users may begin to manually perform windows management operations (e.g., minimizing, moving, resizing) when it reaches a stage called "window thrashing". These techniques focus on defining the "window thrashing" stage and automatically performing window management for users. Unfortunately, these techniques treat a window as an atomic operational unity that cannot efficiently utilize smaller chunks of information contained within windows.

Yeh et al proposed Sikuli (Yeh, Chang and Miller 2009), a scripting environment that allows users to write scripts that reference screenshots of particular controls to refer to existing application elements. To use Sikuli, users first take a screenshot of a widget or an area on screen. These screenshots could afterwards be used as keywords for defining tasks. Python is fully supported as the scripting language and an editor was developed to help the writing of scripts. To perform operations, users can call some functions provided by Sikuli (e.g., Click, Find, Inside), as well as Python's built-in libraries to simulate / trigger user inputs (i.e. mouse and keyboard events). The main applicable area of Sikuli is for normal end users to create custom automatic operations. For example, to minimize all active windows, the two lines of scripts work (Figure 2.4). Note that Sikuli allows users to specify a similarity for image searching and pattern matching, so it can achieve some flexibility.

Figure 2-4 Minimizing all windows in Sikuli

Going further in the direction of image pattern matching, Dixon and Fogarty's Prefab (Dixon and Fogarty 2010) examines pixels as they are drawn on the screen to infer which parts correspond to which widgets. It then allows the interception and replacement of these pixels to change the output of a particular interface. Combined with input redirection, it can present alternate software functionality. Prefab depends on the fact is borders of widgets usually have similar patterns. Taking these patterns into a database, Prefab provides awareness of widget positions for programmers. With this information and input redirection techniques, programmers could develop some generalized add-ons in OS-level. For instance, a target-aware pointing technique like Grossman and Balakrishnan's Bubble Cursor (Grossman and Balakrishnan 2005) (Figure 2.5) and Baudisch et al.'s Phosphor (Baudisch, et al. 2006) which shows users' recent manipulations (Figure 2.6), were implemented in Prefab's architecture.

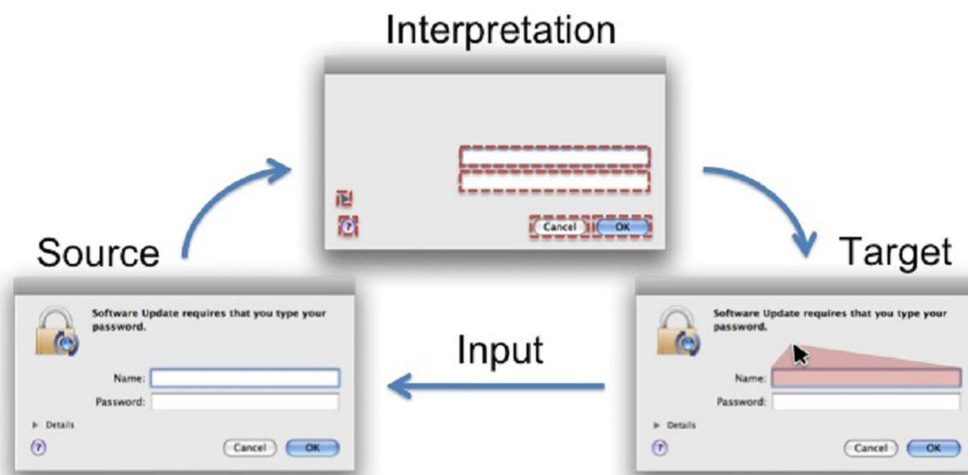


Figure 2-5 Bubble Cursor

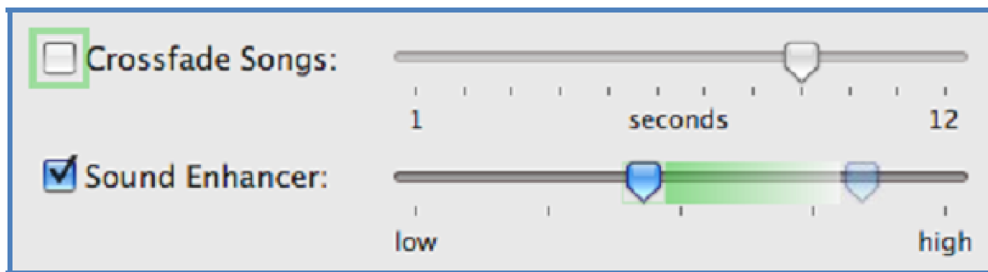


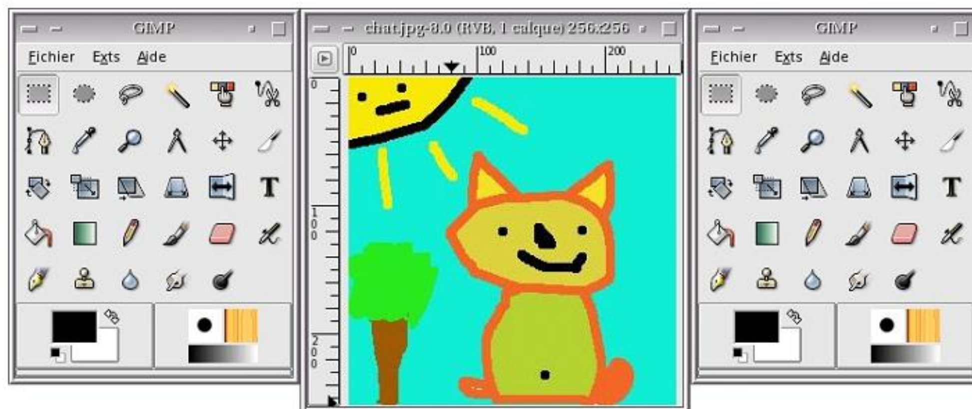
Figure 2-6 Users' recent manipulation histories

Since surface-level modifications do not rely on APIs, they can be fairly widely applied to different applications, program frameworks, or even different operating systems, without much modification of source codes. On the other hand, using this approach, interpreting is usually difficult (e.g. Prefab tries to identify visual widgets, which lay on top of complicated background), since it needs to be trained in particular environments and is easy to be interfered by screen images. More importantly, users are not able to access to data behind screen (e.g. the text in pages out of current view, or widgets that are not in current tab pages). Meanwhile, output is limited to visual elements not in program behavior level. Re-rendering and image analysis may be slow in some cases.

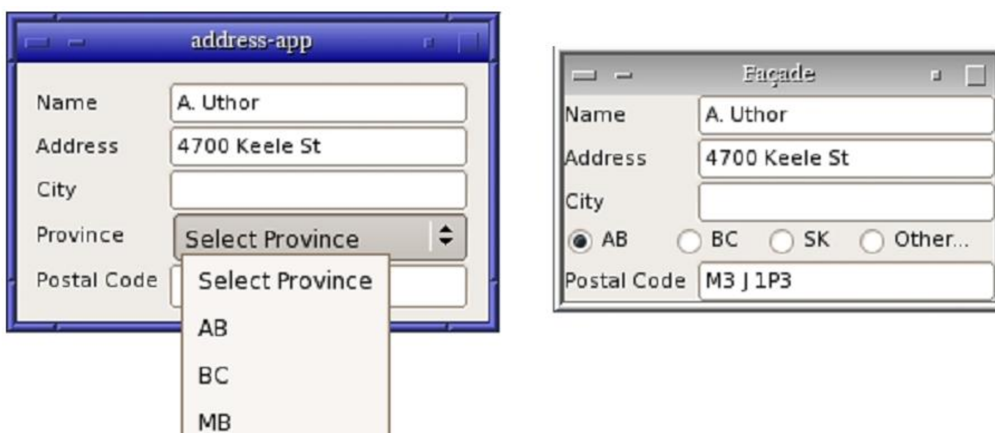
2.2.2 Program Behavior-Level Modification

Unlike the previous approaches, Stuerzlinger et al.'s UI Facades (Stuerzlinger, et al. 2006) intercept individual widgets as they interact with the window server, allowing a developer to replace them at the window server level with an alternate implementation, such as by changing a radio button to a pop-down menu. The Facades system was built based on an X window system called Metisse (Chapuis and Roussel 2005). Metisse, which was designed for both standard daily usage and for support for HCI researches, separates rendering work and interaction processes clearly. Facades system creates a transparent layer on top of window system for window replication and

input redirection. Facades retrieve widget information (e.g. size, position, text, images) through accessibility APIs of GUI toolkits. Using the retrieved boundary information of widgets, Facades can determine the widgets of region of interest specified by users, as well as some visual information of the widgets. An essential component of Facades is FvwmCompositor, a standalone application that merges and composites images to get output widgets or pixels. Metisse provides an off-screen buffer to improve seamless duplication, as well as facilities, for input redirection. Widget replacement of the original application was enabled by APIs of GUI toolkit. Scenarios of Facades include duplicated toolbox and widget replacement (Figure 2.7). Although it uses the accessibility APIs to enable widgets duplication and merging, Facades does not explore the possibility of changing program behaviors.



(a) Duplicated Toolbox



(b) Widget Replacement

Figure 2-7 Facades

Edwards et al.'s SubArctic toolkit (Edwards, Hudson, et al. 1997) extends Java's AWT to provide explicit hooks that allow third-party developers to add new UI modifications. In AWT framework, platform-specific implementation of built-in graphic objects provides similar appearance and behaviors on different platforms. AWT allows the same application codes to run on different operating systems as long as they support JAVA and have AWT installed. Applications use subclasses that derive from basic AWT's graphics objects, of which the APIs provide drawing methods. Within SubArctic's framework, these drawing methods are overridden in subclasses in order to modify output appearance. These hooks provide specific support for extensibility, allowing a third-party developer to add new functionality to existing applications built with the SubArctic toolkit. Although it modifies applications in a program behavior level and touches the codes behind application surfaces, SubArctic's approach focuses on transforming how widgets are drawn; it does not provide explicit support for changing behaviors of program, e.g., adding new functions.

Begole proposed Flexible Java Applets Made Multiuser (JAMM) (Begole 1998), which enabled deeper manipulation of Java classes by swapping classes during Java's serialization streaming for both collaboration-transparency and collaboration-aware applications. It is based on object-oriented replication, where multi-user extensions dynamically replace target user interface objects. Original application providers need not be aware of this replacement. Partially or completely replacing behaviors of existing classes is enabled by this approach. However, this approach only supports serializable classes that do not have dynamic modification after serialization, and it is not safe to replace classes that are already subclassed in original applications.

Besacier and Vernier (Besacier and Vernier 2009) used a similar approach to extend windows management by inserting an immediate layer between applications and system libraries. For example, *CreateWindow* function is called when a user interface

window is created, and *DestroyWindow* function is called when it is closed. In this approach, applications' requests are redirected to a *DiamondSpin* method. This method then provides APIs for third party developers to hook their modifications or functions. The architecture of this approach is shown in Figure 2.8. Besacier and Vernier demonstrated this approach by adding rotation, peeling-back, stacking, zooming, and duplication capabilities to regular windows. This kind of approach – creating standalone libraries that build a wrapper on top of existing GUI libraries – requires a huge amount of effort to explicitly rewrite all functions to support needed custom styles. As mentioned in this paper, some thirty win32 functions take about 5000 lines of codes.

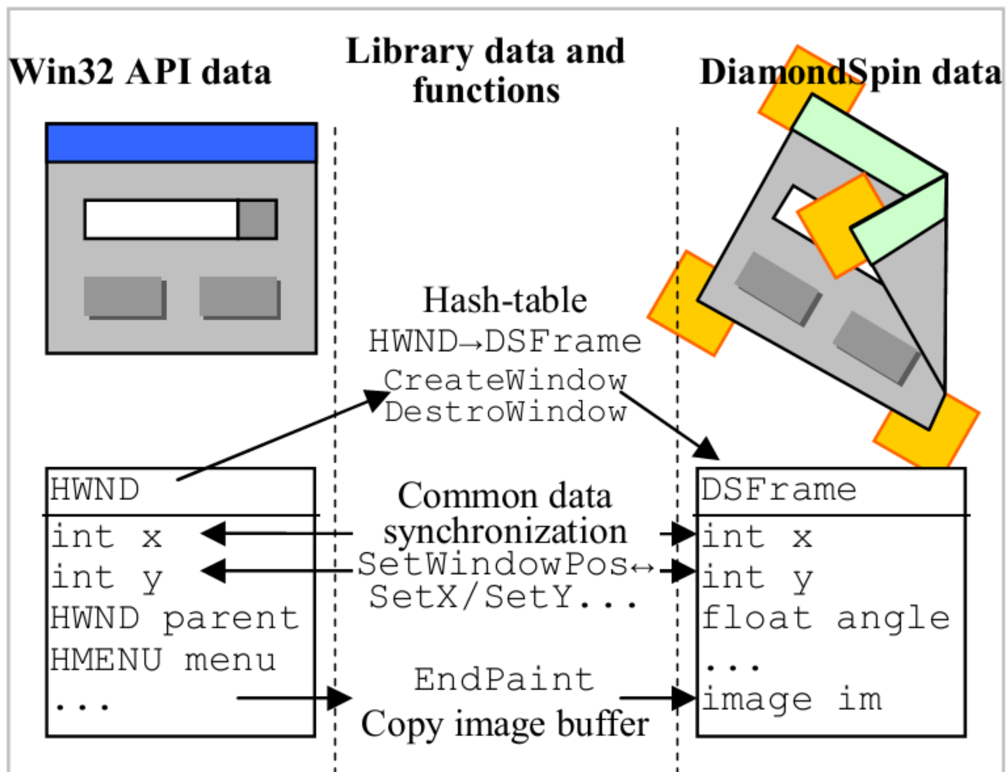


Figure 2-8 Extending the window management using DiamondSpin

Eagan et al.'s Scotty (Eagan, Beaudouin-Lafon and Mackay 2011) system uses injection to perform runtime toolkit overloading, in which an existing toolkit is altered

specifically to provide explicit support for third-party modifications. It provides a *meta* toolkit for developers to modify third-party applications, as well as tools for these developers to inspect existing applications. Eagan proposed runtime toolkit overloading model as a general solution to develop add-ons for third party software. This model contains six components:

- Window and Widget Hooks: Needed to interpret and modify widgets or windows before they are rendered. This kind of operation, for example, includes changing attributes and layout, adding and removing widgets, and minimizing a window. Hence, a hooking mechanism should be provided to access applications.
- Event Funnels: Except for the appearance of windows, a metaclass is also needed to intercept, process, and dispatch events (e.g. mouse, keyboard). Developers could insert their callback functions in the metaclass, so that they can manage all user events.
- Glass Sheets: Glass sheets are a transparent overlay on top of applications; they allow developers to display contents without interfering with applications.
- Dynamic Code Support: The environment should be able to dynamically load developers' modifications into applications and execute them; i.e., modifications are in the form of dynamic add-ons or scripts.
- Object Proxies: Object proxies allow developers to override, overload, or add new methods to particular object instances. This provides the ability to change a program's behaviors.
- Code Inspection: For deep medications that require thorough understanding of original programs, some toolkits (e.g. a hierarchy browser of widget tree) are helpful code inspection.

The prototype software Scotty was implemented in Python using Python/Objective-C bridge, targeting applications that run on Cocoa GUI framework of Mac OS X. Despite the large modification ability enabled by Scotty and the formal identification of the problem, reconfiguration or building of new interfaces within Scotty's architecture (e.g., change colors, texts, and hide items) is completely coding-based. Modern GUI editors have significantly simplified the process of design and developing GUI; this approach is functional but very tedious since GUI editors are not available in Scotty.

2.3 Summary

Much existing software does not support add-on architecture or only supports very limited add-ons, due to additional significant overhead of software design and maintenance. Even for the software that supports fully functional add-ons, the development of add-ons is usually not mature, since third party developers cannot use GUI editors to help implement UI modifications.

Some previous research has focused on providing general add-on architectures for third party software. Within all the research, Scotty's approach, which allows third party developers to build add-ons, provides the most flexibility and power. However, Scotty's approach does not support a WYSIWYG editor (Shneiderman 1993) for developing GUI widgets, meaning that developers have to write text codes to define and set the properties of the GUI widgets that they want to create. Previous studies have proved that interactive building techniques display ten times the effectiveness provided by coding (Myers and Buxton, Creating highly-interactive and graphical user interfaces by demonstration 1986, Hutchins, Hollan and Norman 1985, Myers and Rosson, Survey on user interface programming 1992).

This thesis proposes WADE, which can not only allow developing add-ons for third party software, but also provide a WYSIWYG environment for GUI editing. Table 2.2 presents a comparison between some of previous approaches and WADE.

	Façade	**Prefab**	**SubArctic**	**Scotty**	**WADE**
Surface-level changes	✓	✓	✓	✓	✓
Access to widgets	✓	✓	✓	✓	✓
Changing widgets	via I/O redirection	via I/O redirection	✓	✓	✓
Changing host program			Partially	✓	✓
Modifications are safe and robust	✓	✓	✓		
WYSIWYG	✓	✓			✓
IDE Support					✓
Target User	Novice & Expert	Novice & Expert	Expert	Expert	Novice & Expert

Table 2.2 Comparison between previous approaches and WADE

Chapter 3 Proposed Approach

3.1 Introduction

We present **WADE**, a WSYWYG add-on development environment, and its utility add-ons that significantly ease the task of modifying GUI-based functions in existing software, while still enabling add-on developers to make significant changes to the software behavior. WADE enables novice users to trivially reconfigure and integrate existing add-ons to the host application, even when such functionality is not natively supported. Furthermore, add-on development is greatly simplified through the use of a GUI editor and the IDE.

The WADE prototype presented in this paper works on existing *Windows Forms* applications. A WADE dynamically-linked library (DLL), which is called Injected Add-on Manager, is first injected into the host program, regardless of whether or not it supports add-ons. This DLL retrieves the GUI hierarchy of the host program and communicates it to the IDE Add-on Manager. The latter manager translates this information into declarative language that enables easy modifications through a GUI editor. For straightforward property changes (e.g., changing the position or, appearance of UI elements), a third-party developer can make these modifications directly in a WYSIWYG editor. For more complex modifications (e.g., adding new functions), the editor provides scaffolding to directly associate event handlers to existing widgets. These changes are then written out to a new DLL, which can be injected back to the host program during subsequent invocations.

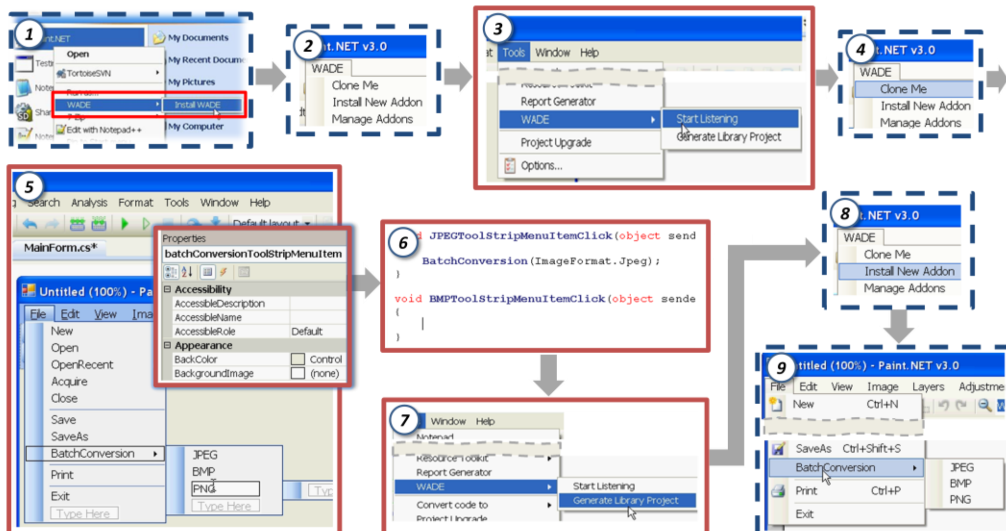


Figure 3-1 Steps of adding a Batch Image Conversion add-on to Paint.NET.

Figure 3.1 illustrates how WADE is used to add a *Batch Image Conversion* add-on to the *Paint.NET* image editing application. The Batch Image Conversion utility converts a batch of image files to a pre-specified format, optionally allowing the user to resize and rename the images during conversion. Steps (1), (2), (4), (8), and (9) are on Paint.NET, while the remaining are on the WADE IDE. (1) The WADE DLL is injected onto Paint.NET. (2) Paint.NET with the WADE menu item. (3) The IDE add-on manager is invoked on the WADE IDE through the “Start Listening” command. (4) The IDE add-on manager communicates with the injected DLL to clone Paint.NET when the “Clone me” command is invoked on the application window. (5) Once Paint.NET is cloned onto the WADE IDE, GUI widgets can be modified directly using the GUI editor, which also generates event handler templates (6). (7) The Batch Image Conversion add-on code is compiled to generate the DLL (8), which is linked with Paint.NET at runtime (9).

Note that this functionality is beyond the reach of surface-level methods such as Prefab (Dixon and Fogarty 2010), as it requires access to the underlying program structure and requires learning at least part of the program's organization with Scotty

(Eagan, Beaudouin-Lafon and Mackay 2011). More generally, when the source code / API is unavailable, manipulating a GUI framework for reconfiguring or integrating add-ons is challenging. By gaining access to the host's GUI hierarchy and converting it to an editable form, WADE enables (i) a user with little programming knowledge to easily reconfigure GUI components, even if such capability is not originally supported by the host application; (ii) add-on developers to modify program behavior in a WYSIWYG fashion; and (iii) HCI researchers to evaluate novel interaction techniques on existing popular applications in real-world settings. In addition, WADE also provides a number of utility add-ons, including a stand-alone property editor for reconfiguring the host program without the use of the IDE, and a generic user-interaction logger.

3.2 System Architecture

Figure 3.2 presents an overview of the WADE architecture. WADE consists of an Integrated Development Environment and a number of utility add-ons. In this section, we focus on the WADE IDE. The WADE IDE has two components: the Injected Add-on Manager and IDE Add-on Manager. The injected add-on manager is injected into an application's add-on address space using DLL injection techniques (Berdajs and Bosnic 2010, Richter 1994, Pietrek 1994, Kuster n.d., Pulley n.d., Working with the AppInit_DLLs registry value n.d., Newcomer n.d., CrankHank n.d.). The functions of the injected add-on manager are two-fold: (i) to retrieve properties of UI widgets from target applications and send them to the IDE using socket-based communication and a file cache on the disk, and (ii) to load any compiled add-on onto the application's address space and invoke the add-on's initialization method.

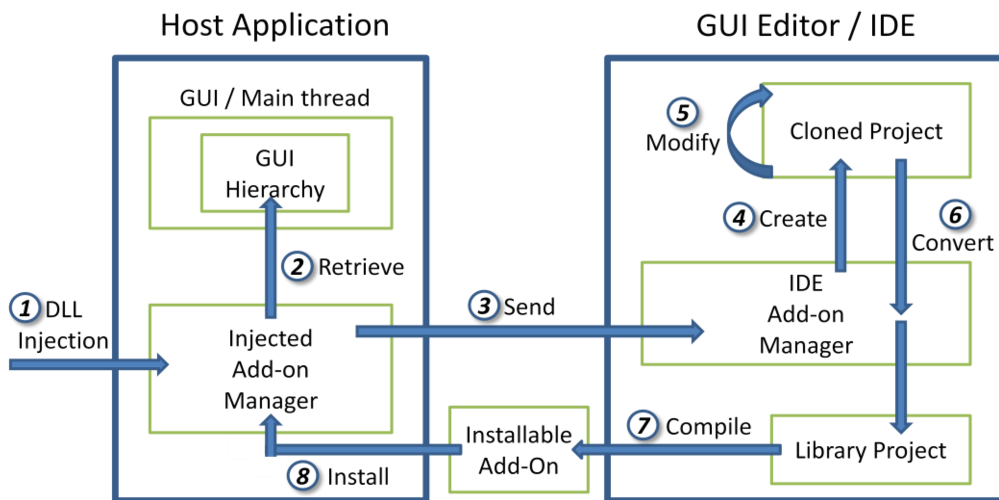


Figure 3-2 Architecture overview of WADE

The IDE add-on manager also has two main functions, namely, (i) to receive UI properties sent by the injected add-on manager and create a project with the application's cloned interface, and (ii) to generate an add-on DLL from an add-on library project. This add-on can then be loaded onto an existing application at runtime.

Integration of an add-on to a third-party application is accomplished in the following manner:

- WADE first uses DLL injection to load the (injected) add-on manager into the host application at runtime. Because the add-on manager runs in the host application's address space, it has both read and write access to the UI component hierarchy.
- WADE then retrieves the UI component hierarchy and serializes it to IDE add-on manager.
- WADE creates a clone of the host application in the IDE, enabling third-party developers to modify the cloned application interface using the IDE's GUI editor.

- Once all changes are completed, WADE analyzes the changes made to the cloned project and writes these changes into a DLL file that can then be loaded back into the application by add-on loader.

The WADE IDE itself was developed as a plugin to the SharpDevelop IDE and implemented for Windows Forms applications on the .NET Framework, running on the Windows operating system. In this section, we introduce some fundamental concepts necessary to modify applications in the .NET framework. In a later section, we compare these approaches to those available in other environments.

3.3 Runtime Intervention through DLL injection

WADE facilitates the creation of add-ons, such as those that change their appearance and behavior at runtime, to third party applications. This ability requires access to the application's interface objects. There are two primary ways to gain such access: (i) directly manipulating the binary executable of the host application, or (ii) creating additional helper libraries to intervene in the host application's behavior within its runtime processes. The former method is both difficult and risky, and thus carries with it a high possibility of causing crashes. WADE instead adopts the second approach.

To intervene in the runtime processes, there are again two possible approaches: employing OS level system calls, or injecting code into the processes space. Since the OS typically provides only a limited number of system calls (e.g., kill), it cannot fulfill the diverse requirements for add-on development. Thus, we choose code injection to achieve our goals. Various code injection approaches are possible for different operating systems. Some of these include monitoring the communication between the app and window manager (e.g. Facades (Stuerzlinger, et al. 2006)), modifying the toolkit to support new functionality (potentially requiring all apps to be re-linked, e.g.

Mercator (Edwards, Mynatt and Stockton, Providing access to graphical user interfaces —— not graphical screens 1994)), replacing shared libraries (e.g. WINE (Besacier and Vernier 2009)), using scripting/design hacks (e.g., Input Managers (Eagan, Beaudouin-Lafon and Mackay 2011)), Scripting additions, using magic Registry keys (e.g. WADE)), or using kernel hacks (e.g. CreateRemoteThread). In WADE, both registry key and kernel hacks techniques are implemented.

The Registry Key and CreateRemoteThread methods enable WADE to run some external codes (named *BootStrap.dll* in WADE) as a thread in the host application's address space. However, the *BootStrap.dll* is written in C++. C++ is used for the purpose to explicitly specify a function to be called automatically when the library is loaded. In C++, we can easily use *DllMain* to achieve it (Heege 2007, Wallach 2000), but in C#, the programming language used by target applications, there is no such mechanism. Hence, we need a C++ based DLL for using global hooking and a C# based DLL (named *Injectee.dll*) to do the remaining work.

The .NET framework has two main components: the Common Language Runtime (CLR) and the .NET framework class library. The CLR is the foundation of the .NET framework. The runtime can be regarded as an agent that manages code at execution time, providing core services such as memory management, thread management, and remoting, while also enforcing strict type safety and other forms of code accuracy that promote security and robustness. In fact, the concept of code management is a fundamental principle of the runtime. Code that targets the runtime is known as managed code, while code that does not target the runtime is known as unmanaged code. Figure 3.3 shows the relationship of the common language runtime and the class library to applications and to the overall system. The .NET framework can be hosted by unmanaged components that load the common language runtime into their processes and initiate the execution of managed code, thereby creating a software en-

vironment that can exploit both managed and unmanaged features (Network n.d.). Figure 3.4 shows how to load a CLR and call managed codes in the CLR using C++ (How To Inject a Managed .NET Assembly (DLL) Into Another Process n.d.).

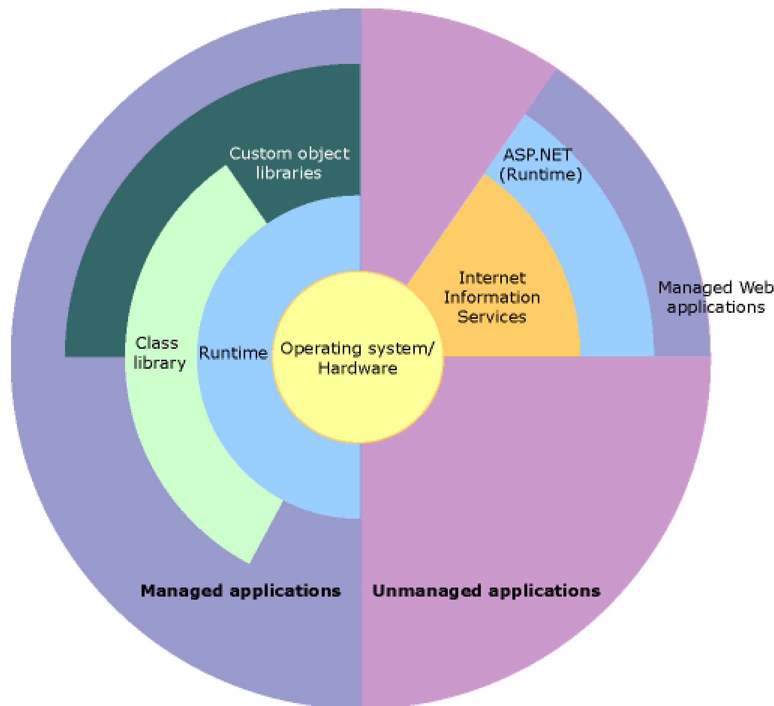


Figure 3-3 Common Language Runtime in .NET framework

```
// Loading Injectee.dll
ICLRRuntimeHost *pClrHost = NULL;
HRESULT hr = CorBindToRuntimeEx(
    NULL, L"wks", 0, CLSID_CLRRuntimeHost,
    IID_ICLRRuntimeHost, (PVOID*)&pClrHost);

// Push the big START button shown above
hr = pClrHost->Start();

// Okay, the CLR is up and running in this (previously native) process.
// Now call a method on our managed C# class library.
DWORD dwRet = 0;
hr = pClrHost->ExecuteInDefaultAppDomain(
    injecteePath,
    L"Injectee.Initialization", L"Init", L"Parameter", &dwRet);

// Optionally stop the CLR runtime (we could also leave it running)
hr = pClrHost->Stop();

// Don't forget to clean up.
pClrHost->Release();
```

Figure 3-4 Creating CLR using C++

3.4 Modifying GUI properties

To modify elements in the GUI thread, an application typically uses call back functions to allow worker threads to update results with UI threads. However, direct update of the UI thread by worker threads is typically not allowed as different threads may not be aware of each other. Trying to update the same UI component concurrently can cause unpredictable behavior. Therefore, a centralized managing mechanism, such as an event queue, is typically employed to avoid this problem. In WADE, our codes run in a separate thread created by the injected DLL. Therefore, it also needs to use call back functions in order to modify elements in the GUI thread. In Windows, this is achievable using asynchronous callback, a windows-specific event queue implementation. The .NET framework provides the *Invoke* method to access the UI thread under such scenarios:

Invoke(): This method allows dispatching of a method on the current UI thread and provides the basic tools for runtime application modification.

In Object Oriented programming, widgets are described as classes; the properties of the UI are stored as instance variables inside these classes, which can often be modified by calling the corresponding getter and setter methods. To modify GUI properties, a developer just needs to obtain read and write permission of these class instances, which is automatically granted to the injected DLL within the host application at runtime.

Once the basic principles of runtime GUI modification are understood, how individual operations (such as modification, addition, and deletion of widgets) are implemented in WADE can be addressed. There are three primary operations involved in modifying an existing application's interface: adding a new widget, deleting an existing widget, and modifying the properties of an existing widget.

3.4.1 Retrieving GUI Information

GUI frameworks typically organize widgets into a forest of trees. Each tree in the forest represents the widgets that belong to a particular window. In order to gain access to all of the widgets in an application's interface, it is sufficient to get access to the root of each of these trees and to perform a tree walk to enumerate the structure and properties of each of the widgets in the hierarchy.

The root of each tree is thus typically a Window widget. WADE uses the *System.Windows.Forms.Control* class in .NET, whose Controls property exposes a collection of all of these child controls. Through this component, we can access the structure and properties of an entire application's existing interface.

3.4.2 Modification and Addition

Modifications of widget properties and the addition of new GUI widgets are both straightforward. Modification is simply achieved by using the getters and setters of the widget instance object to modify its properties. Widgets can be added by creating a new widget instance at runtime and attaching it to its parent widget.

3.4.3 Deletion

Deleting widgets is more complicated. While it is straightforward to use the *Dispose* method provided by .NET to remove widgets objects at runtime, this method may have unforeseen consequences due to unknown runtime dependencies to these widgets. As such, deleting widgets can be risky, potentially resulting in an application crash. Therefore, instead of deleting widgets, deletion is simulated by making them invisible.

3.4.4 Modifying program behaviors

Except for re-configuration (modifying appearance of UI widgets), adding new functions into existing applications often requires associating programming logic code with GUI elements. This is often achieved using event handlers in modern GUI frameworks. Event handlers need to be specified by developers, who describe the actions/behaviors that will happen after attaching the event. To change the functions of existing applications, the developer can detach original event handlers, and attach his own.

3.5 Supports for the GUI Editor

Through the injected DLL at runtime, third-party developers can modify the appearance and behavior of the host application without accessing its source code. However, GUI modification and add-on development via pure scripting can be tedious and inefficient. For example, the simple task of adding a new Menu Item into an existing Menu requires the developer to first identify the name and position of the menu item. In the absence of visual aids, this can be a lengthy trial and error process even for experts.

The same modification task can be significantly simplified using a GUI editor, which is often provided by many modern IDEs such as Microsoft Visual Studio, Eclipse, NetBeans, etc. GUI editors provide a WYSIWYG style of GUI modification, and support automatic creation of an event handler skeleton code to the GUI widgets, making association of program logic with GUI components much easier to handle. WADE, built on top of the SharpDevelop IDE, provides third-party add-on developers with such an environment.

Nevertheless, GUI editors in existing IDEs are designed to facilitate the creation of new interfaces from scratch, rather than to modify existing interfaces. Furthermore,

the code associated with the GUI components of the original program is not available. To solve this problem, the GUI hierarchy needs to be imported into an available GUI editor for the third-party developer to modify, and, more importantly, to then apply the modifications back to the host application authentically.

SharpDevelop, an open source IDE mainly for developing .NET and Mono (an open source implementation of .NET) applications, was selected for the WADE prototype, because of its well-designed architecture and abundance of open source add-ons, which make implementation convenient. We implemented WADE based on SharpDevelop by writing an IDE add-on manager, which itself is an add-on of SharpDevelop. To facilitate add-on development in SharpDevelop, the IDE add-on manager should be responsible for inter-process communication, creating project in IDE, and code conversion.

3.5.1 Inter-Process Communication

First, the Injected Add-on Manager walks through the widget forest, retrieves the properties of each widget in the hierarchy, and sends this information back to the IDE Add-on Manager. Since the Injected Add-on Manager and IDE Add-on Manager are running in separate processes, inter-process communication techniques are needed to transfer GUI information from Injected Add-on Manager to IDE Add-on Manager. There are several common solutions for this (Interprocess Communications (Windows) n.d.).

- **Cache File**. File is a block (it may or may not be physically continuous) of binary information stored on hard disk. A file can contain texts, images, sound, or other custom types of data. Intuitively, WADE can save GUI information in a cache file (or files) on hard disk to fulfill requirements. Most operating systems natively provide both user commands and programming

interfaces for creation, reading, writing, deletion, setting file attributes and other file operations. To access a file, enough permission is required. For example, in our case, Injected Add-on Manager needs write permission in create and write files in a directory, and IDE Add-on Manager only needs read permission to the files. In most situations, applications are able to find some directories provided by systems (e.g., current user's home folder) where they have full access or can create a new folder, in which they have full access under given directory. This is the first advantage of *cache file* solution: it is easy to implement in a wide variety of platforms and operating systems. As another advantage, although limited to many factors (e.g., type of file system, hard disk's free space, users' personal free space), the available space for applications typically exceeds hundreds of megabytes, which is more than enough for WADE that only need tens of megabytes. On the other hand, the most significant drawback of the *cache file* solution is that IDE Add-on Manager does not know when cache files are ready to be read. One complementary means is to write the current state (e.g., ready for reading, ready for writing, locked) just inside the cache file as a header. Then WADE can work as shown in Table 3.1. This solution requires an additional monitor for IDE Add-on Manager to periodically detect the state of the cache file. Meanwhile, creating numerous small files to transfer information is not efficient due to the limitations from current mechanical performance of hard disks.

Cache File State	Action for Injected Add-on Manager	Action for IDE Add-on Manager
Ready for Reading	Wait until reading done	Read GUI information
Ready for Writing	Write GUI information if any	Wait until writing done
Locked	Wait until unlocked	

Table 3.1Cache file solution

- **Shared Memory**. Since mechanical performance is the performance bottleneck of a *cache file* solution, a memory based approach could possibly overcome it. Shared memory is a block of memory that can be accessed by multiple programs. It is supported by many operating systems (e.g. Unix, Windows), languages (e.g. C/C++, PHP), and libraries (e.g. Qt, Boost). Due to faster access of RAM, *shared memory* solution could be more efficient for data communication than cache files. However, compared with a *cache file* solution, shared memory directly manipulates data in memory level, which is unnatural for a high level object oriented programming language, as well as more error-prone.
- **Clipboard**. As a special example of shared memory, clipboard is an OS-scope central shared memory that can be accessed by applications. It is easier to use because programmers do not need to explicitly apply for a block of shared memory. Side effects include losing some flexibility and being exposed to all other applications.
- **Pipeline**. Pipeline is a method widely used in many modern systems for data communication, notably for command line applications in Unix-like systems. Pipeline involves chaining a set of processes by redirecting their standard streams, so that the (standard) output of a process becomes the (standard) input of its next one. For GUI applications, some system calls enable programmatically using of pipelines. This convenient approach of inter process data communication, however, does not fit WADE's requirement, because using Pipeline to transfer GUI information may interfere with the normal input / output of applications.
- **Signal**. Unlike Pipeline, which focuses on data communication, Signal creates notifications between processes. A process can send to and receive

signals from operation system or other processes. If a signal handler is defined, the receiver process of a signal executes the handler; otherwise, a default handler is executed. *Signal* is a very mature way to solve the problem of requiring a background monitor which exists in *Cache File* solution, since the OS will notify IDE Add-on Manager when necessary. However, this method is not able to carry data and is limited to the pre-defined signals provided by OS.

- **Socket**. Socket is another popular way in which inter-process data flow. It is typically used based on the Internet Protocol for communication between different computers. A transport protocol (e.g., TCP (Stevens and Wright 1994), UDP (Postel 1980)) controls data transmission. Socket is also applicable for communication between local processes, or those with a specified local IP address. A significant advantage of the *Socket* solution is that for a large amount of small objects, socket communication is more efficient than cache files on hard disk. Additionally, it is easy to separate Injected Add-on Manager and IDE Add-on Manager in different computers, if necessary.
- **Remote Procedure Call (RPC)**. In all of the previous solutions, except for the *Signal* one, the periodical monitor is required to detect updates from Injected Add-on Manager. RPC is another way to achieve our goal without using a monitor. RPC allows a program to execute a procedure / subroutine in another running application (address space) (Birrell and Nelson 1984). It can also be achieved via a network. The implementation of RPC, however, is not as standardized as the *Socket* approach. It might not be easy to implement WADE in different frameworks and OSs if RPC is adopted. The Injected Add-on Manager and IDE Add-on Manager may have to use the same programming framework, which decreases the flexibility.

WADE adopts a combination of *Cache File* and *Socket* as a solution, based on the fact that cache files are large enough and easy to use; meanwhile, sockets are efficient at transferring small objects and performing cross-framework and cross-network. When requested by users, IDE Add-on Manager creates a TCP listener for local TCP streams and starts a new thread to periodically check for new messages. Injected Add-on Manager correspondingly connects to the TCP server and builds a network stream. As discussed in the section *Retrieving GUI Information*, the Injected Add-on Manager then traverses the widget forest of the target application and retrieves necessary GUI information. For most widgets, Injected Add-on Manager retrieves widget name, size, location, text, etc. and directly sends them using sockets. For widgets that have background images, the images are saved to cache files. Some container widgets (e.g., *Menu*, *ToolStripMenu*) have relatively complicated structure, so they are also saved to cache files in XML format.

3.5.2 Creation of Project in IDE

After receiving complete GUI information from Injected Add-on Manager, the IDE Add-on Manager then builds a project with the same UI properties extracted from the original program. With the extracted UI information, the IDE add-on manager *clones* the existing interface into a new project in the IDE. This step is generally feasible in many platforms and programming frameworks, since many modern IDEs (e.g., SharpDevelop, Visual Studio, and Eclipse) provide API for their add-ons to programmatically create projects and add new (UI) components. Meanwhile, creating UI widgets in a project in IDEs usually involves modifying some parts of source codes. The mapping from creation of UI widgets to the corresponding changes in source codes is open to public; thus, to programmatically create or modify UI widgets of projects, we can also directly modify source codes of projects.

WADE uses the former approach – calling APIs provided by SharpDevelop to create and set properties of UI widgets. SharpDevelop adopts a singleton design pattern (Jahnke and Zundorf 1997). It provides a static *WorkbenchSingleton* class, which contains some static members, e.g., *Workbench*, *MainWindow*, and *StatusBar*. As shown in Figure 3.5, *WorkbenchSingleton.Workbench* contains a member *Active-ViewContent*. If the current project is a Windows Forms project, and the IDE is in a GUI design view (as in Figure 3.6), the *ActiveViewContent* should be an instance of *FormsDesignerViewContent*, which contains an instance of *IDesignerHost*. *IDesig-nerHost* is an interface provided by .NET framework to allow developers to manage designer transactions and components when building custom design-time behavior. This is the essential part of the GUI editor of SharpDevelop. The IDE add-on manag-er of WADE also uses the *IDesignerHost* to create and manage UI widgets. This in-terface provides a *CreateComponent(Type widget_type, String widget_name)* method to create a component of the specified type and name, and adds it to the design docu-ment. This method returns the created widget of type *IComponent*, the fundamental base interface of all Windows Forms widgets.

To modify the properties of newly created widgets, the *IComponent* instance is cast to the type of target widget. Then new properties are directly set in the instance, or the .NET reflection technique can be used to perform a generic setting. Using UI hie-rarchy information sent from Injected Add-on Manager, the IDE Add-on Manager creates a project that has replicated UI widgets. This step is called “cloning” in this thesis.

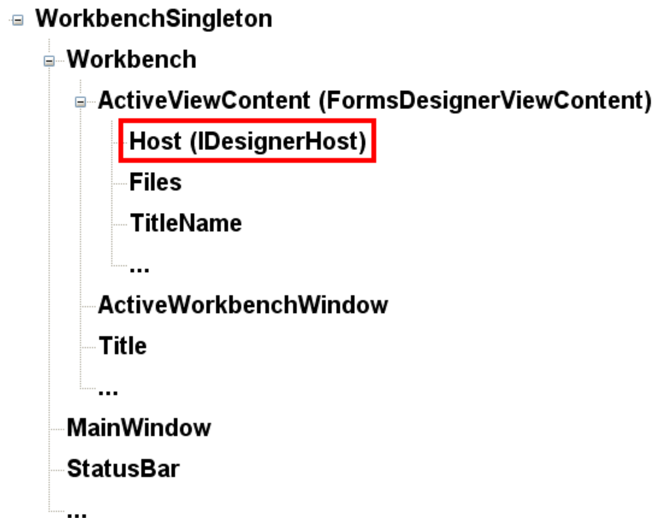


Figure 3-5 SharpDevelop class hierarchy

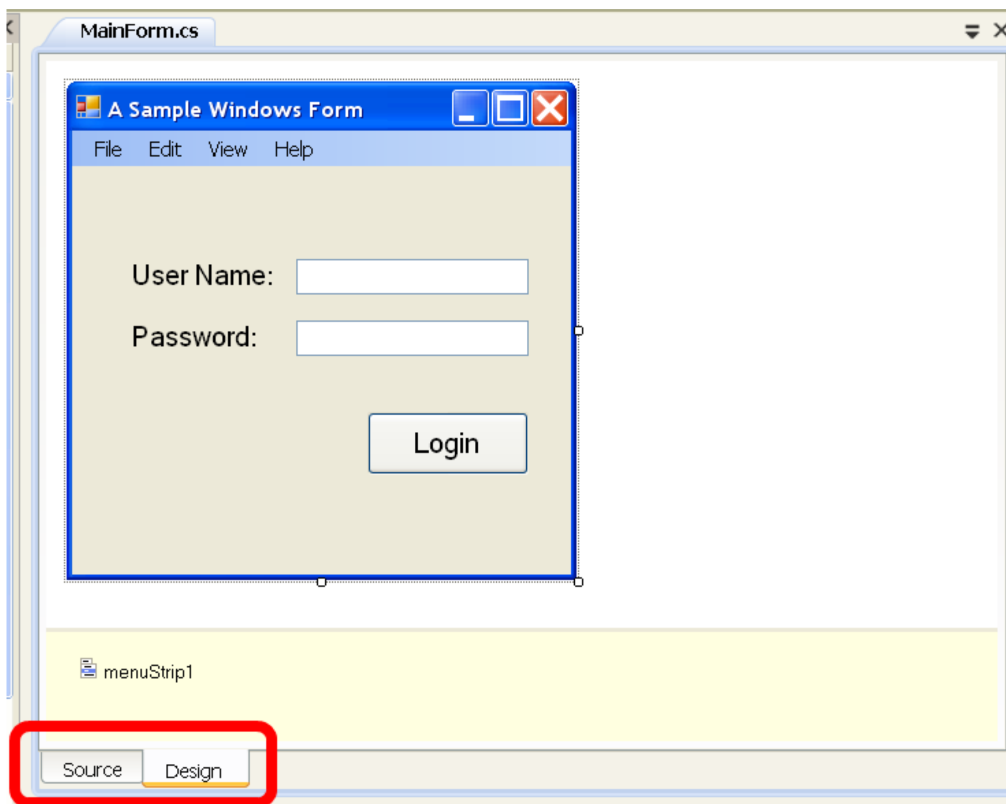


Figure 3-6 SharpDevelop design view

3.5.3 Code Conversion

After cloning, an add-on developer can directly modify the GUI elements via the GUI editor. Once modified, it is necessary to apply these arbitrary modifications back to the host application. However, the edited project cannot be directly compiled to obtain the expected library due to some issues. These issues and corresponding solutions for a Windows Forms library are discussed below.

Several modifications should be applied at the project level. Since SharpDevelop does not support the use of GUI editor for a Windows Forms library project, in order to enable the GUI editor, the type of the cloned project must be a standalone Windows This means that the outcome of compilation is an executable Windows Forms application, not the expected Windows Forms library. Thus, the first modification is changing the output type of the project to library. Next, a *Program.cs* file in Windows Forms project, as shown in Table 3.2, is used for launching the main window. This file is neither useful nor legal for a library project, so it is excluded from the project. The functions of *References*, *AssemblyInfo.cs*, and *MainForm.resx* are the same in standalone Windows Forms applications as in libraries; thus they can remain unchanged.

Component Name	Function
References	I.e. libraries that provide services to projects, including system built-in references and user references
AssemblyInfo.cs	Defines some information about output assembly, e.g. company name, copyright, version no., etc.
Program.cs	Contains Main function, which will open MainForm
MainForm.cs	Is a partial definition of MainForm. Usually developers' manual coding is in this file
MainForm.Designer.cs	Another part of main form's definition. Usually codes generated by GUI editor are in this file.
MainForm.resx	Contains resources of the project, e.g. images

Table 3.2 Windows Forms project files

When developers finish modifications in an original project and click *Generate Library Project* menu item, WADE copies the current project, and creates a duplicated one. WADE does not destroy the original project, thus, developers can revise the original one using GUI editor. WADE performs these conversions in the newly created duplicated project.

More conversion should be done in source code level – the *MainForm.cs* and *MainForm.Designer.cs*. Before discussion of the actual conversion, a good understanding of the source code structure of Windows Forms is critical. In Windows Forms, the definition of MainForm class is divided into two parts: *MainForm.cs* and *MainForm.Designer.cs*. Developers usually write codes in the *MainForm.cs*, which typically contains definitions of event handlers for UI widgets and non-visual elements (e.g., data variables and definitions of custom classes). The latter file is auto generated by GUI editor, which is used for describing UI widgets. GUI editor translates *MainForm.Desinger.cs* and then draws UI widgets in design space. It also anti-translates developers' modification of UI widgets back to the file, which typically contains four parts:

- A container variable used to keep track of non-visual components.
- A *Dispose* method that overrides the one derived from Form class.
- Declarations of all UI widgets added by GUI editor.
- An *InitializeComponent* method to manage all UI widgets. This method contains initializations and property settings of UI widgets.

In the converted library project, only the declarations of UI widgets and the *InitializeComponent* method are needed. In the *InitializeComponent* method, like C++, Java, and some other object-oriented programming language, C# uses the keyword *new* to allocate memory and call class constructor to create an instance of an object. In this

step, existing widgets of original applications and newly added widgets of developers need to be treated differently. To identify this point, after the cloning step, WADE saves a backup copy of original cloned *MainForm.cs* and *MainForm.Designer.cs*. For both original source codes and edited source codes, WADE builds a dictionary of all the widget names. The comparison between the widgets from the host application and those from the modified program enables detection of any deletion or addition of widgets. Corresponding conversions are:

- Added Widgets. For widgets newly added by developers, no conversion is needed, since they are declared, allocated, and modified consistently in Windows Forms project or library project.
- Deleted Widgets. For the widgets to appear in a dictionary of original source codes but not in edited codes, developers must delete them in GUI editor or manually. As discussed in the section *Deletion*, WADE supports deletion operation by making widgets invisible to decrease programming risks. The corresponding codes needed include adding and calling a *HideAll* method at the beginning of *Run* method (the equivalent *Main* function in WADE approach), which makes all widgets in the original program invisible at the first. Then for all widgets remaining in edited codes, statements are added to make them visible. This gives the expected result: hiding all deleted widgets.
- Edited Widgets. The widgets that appear in both original and edited codes are the ones that developers want to use to change the properties of the original program. For these widgets, we need to associate edited codes to pointing to existing widgets. The Injected Add-on Manager sends a *ComponentDictionary* to library add-on when invoking its *Run* method. The allocation step in the *InitializeComponent* method should be replaced with an associating ac-

tion, so that edited widgets really refer to the instances in running applications (Figure 3.7).

```
this.button1 = new System.Windows.Forms.Button();
this.button2 = new System.Windows.Forms.Button();

this.button1 = new System.Windows.Forms.Button();
this.button2 = ComponentDictionary["button2"] as System.Windows.Forms.Button;
```

Figure 3-7 Code conversion example one

The above approach, however, does not identify modifications made to existing widgets: it only identifies structural modifications to the widget hierarchy. Therefore, when an add-on is loaded, it updates all properties of the originally-cloned interface, including those that were modified by the add-on developer and those that not.

Another main conversion is that in Windows Forms codes, all properties of MainForm are called using *this* keyword, because all codes are inside the definition of MainForm. However, in library project, MainForm is a standalone widget just like other widgets declared inside the class definition. Thus, the declaration of MainForm should be added. Meanwhile, all changes of MainForm's property should be redirected (Figure 3.8).

```
this.Visible = true;

this.MainForm.Visible = true;
```

Figure 3-8 Code conversion example two

One more code level conversion involves redirecting the project resource. In Windows Forms, resources are defined within *MainForm.resx*, and an instance of *Com-*

ponentResourceManager class is created (Figure 3.9). Because a uniform main class name must be used for loading DLL add-on, the class name should also be changed.

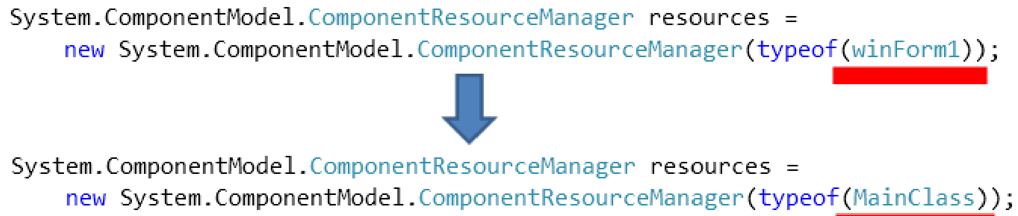


Figure 3-9 Code conversion example three

Conversion for *MainForm.cs* is relatively simple and involves:

- Changing class name to predefined *MainClass*
- Adding declaration of variable *ComponentDictionary*
- Adding main function *Run*, which initializes *ComponentDictionary* using the object passed from Injected Add-on Manager, class *HideAll* method, and *InitializeComponent* method
- Adding definition of method *HideAll*

Upon merging all the aforementioned conversions to code files in the generated project, an add-on that can be loaded into the application must be built. When building the add-on, conversions are compiled into a new DLL and injected back into the host application. Injected Add-on Manager calls its *Run* method to execute the procedure.

Chapter 4 Utility Add-ons

Using the WADE IDE, we aim to simplify development of add-ons for third party software. In this section, we introduce four add-ons we developed using WADE for various purposes, either to enable powerful new functionality or to solve UI related problems for existing applications.

4.1 Property Editor

Although the WADE IDE simplifies add-on development, it requires basic programming skills to operate, and does not solve the problem of allowing end users without programming knowledge to reconfigure their UI. To address this issue, we developed a property editor add-on. The property editor is a generic add-on module that can be installed on any Windows Form-based application (Figure 4.1). Once installed and launched, it displays an editing window and allows users to change basic UI properties such as text label, color, font, size, location, etc. It can also hide widgets. Once a change is made, it applies immediately to the host application. This add-on is implemented simply by associating the property editing widget (provided by Windows Forms) with a widget selected by the user in the host application.

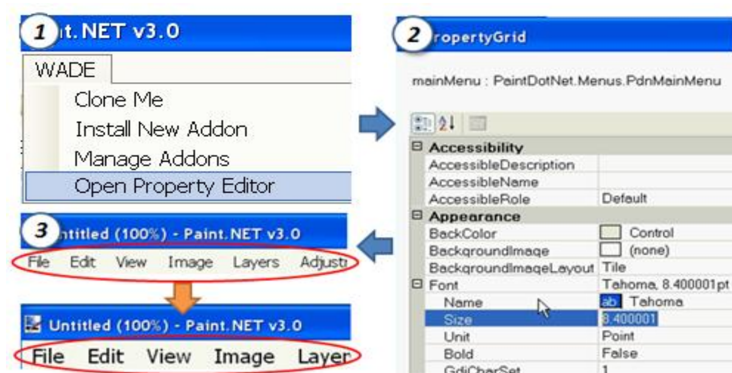


Figure 4-1 PropertyEditor add-on

4.2 Interaction Logger

As previously seen in the usage scenarios, WADE can be used by user interface researchers to perform evaluations involving existing applications. One essential task for any user study is to log user interactions with the application. We therefore developed an interaction logger add-on that can capture all the mouse and keyboard events with the host application, and can be played back at a later time (Figure 4.2). Using this add-on, researchers can easily record, compare, and analyze user interactions and calculate performance-related measures such as timing and accuracy. This add-on is achieved by using a global mouse and keyboard hook, which can detect and log all mouse and keyboard input events on the host application.

Figure 4-2 EventRecorder add-on

4.3 Multi-stroke Marking Menu

Marking menus are efficient, gesture-based menus (Zhao, Agrawala and Kinckley, Zone and polygon menus: using relative position to increase the breadth of multi-stroke marking menus 2006, Zhao and Balakrishnan, Simple vs. compound mark hierarchical marking menus 2004). However, they are not supported by most applications. We implemented a generic multi-stroke marking menu add-on for Windows Form applications. Once installed, users can construct customized marking menus by selecting UI commands from the host application, which can significantly improve their performance for invoking these commands. The marking menu add-on is im-

plemented by integrating an open source version of the code with the WADE architecture (Figure 4.3). The resulting add-on can be installed by Windows Form-based applications.

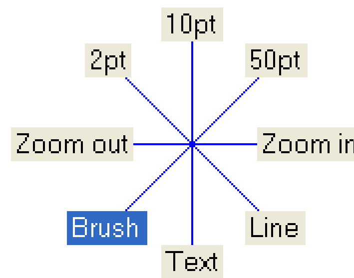


Figure 4-3 MarkingMenu add-on

4.4 Heat Map Generator

One situation that often troubles users is that relevant functionalities for certain tasks are located in different applications or services on the computer. For example, although Paint.NET supports many image editing effects, it does not support drawing heat maps (Wilkinson and Friendly 2009) based on input data. However, such features can be supported by other services or applications (Wikipedia n.d.), but it could be tedious to complete a series of tasks going back-and-forth between the two applications. Using WADE IDE, we merged an existing open source heat map generating library (Media Interaction Lab n.d.) to create a simple add-on that will allow users generate heat maps (Figure 4.4).

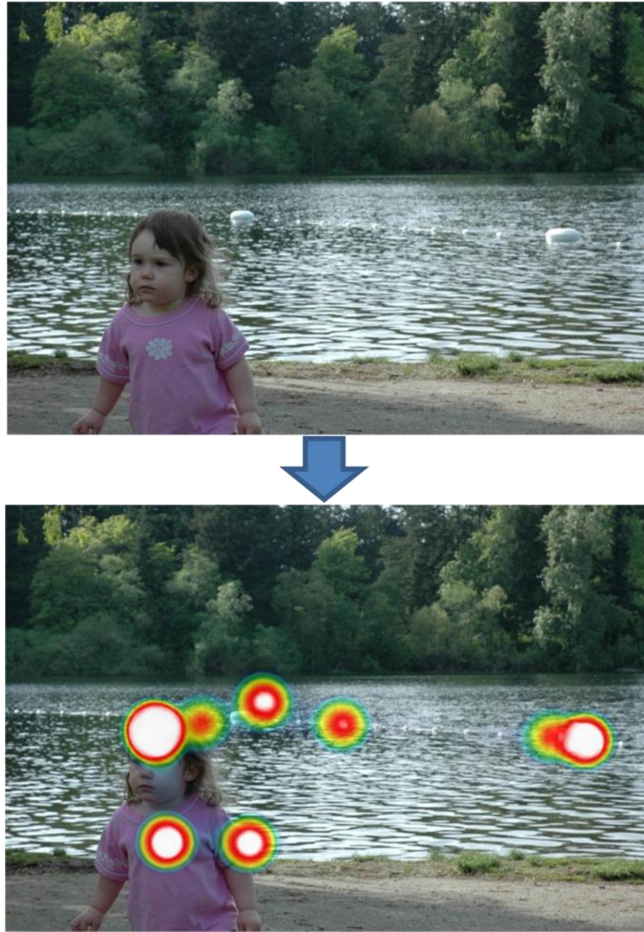


Figure 4-4 HeatMapGenerator add-on

4.5 Summary

Note that, without WADE, creating any of the above add-ons is both challenging and tedious, and even if the add-ons can be synthesized, they are typically application specific. It is much easier to develop generic add-ons using WADE that can be applied to most Windows Form based applications. The first three add-ons described above are generic add-ons that can be directly applied to most Windows Form-based applications. The external call add-on requires the information of the target object, and therefore not directly applicable to other applications. However, it can be easily modified to suit other application's requirement. All the above add-ons were developed by a single programmer within a month.

Chapter 5 User Study

We now present the results of a user-study to demonstrate the efficacy of WADE. As previously illustrated, WADE can benefit three types of users: (i) the end-user with no programming skills who wants to reconfigure existing software without access to its source code, (ii) third-party developers who need to reconfigure or introduce add-ons to third party software, and (iii) user interface researchers who would like to synthesize GUI variants in order to perform a comparison study.

Among these three scenarios, we are unaware of any current tools that can support the first scenario, making it difficult to perform a comparison study. The second and third usage scenarios, however, can be achieved using runtime toolkit overloading approaches like Scotty (Eagan, Beaudouin-Lafon and Mackay 2011). While it is generally expected that WADE's GUI editor and IDE will significantly simplify software modifications, a formal comparison will enable us to appreciate and quantify potential performance gains with WADE, as well as understand the characteristic differences between the two approaches. Therefore, we conducted an experiment to compare WADE with a Scotty-like approach for third party add-on development.

5.1 Tools

In order to build a Scotty environment for tasks, we requested the author of Scotty for its source codes and executable files. However, the author replied that current Scotty is complicated to install since it has messy dependencies. Meanwhile, Scotty is developed for the Cocoa framework in Mac OS, while WADE runs on the .NET framework in Windows. Cross-platform testing may introduce potential influences to

the experiments. As such, we created a Scotty-like development environment (or Scotty simulator) to support the user-study tasks using the following tools:

- **Runtime add-on manager**: a tool that enables a compiled add-on to be installed onto an existing program at runtime.
- **Managed Spy**: a utility program provided by Microsoft to allow developers to spy on an application's GUI at runtime. Figure 5.1 is a screen-shot of the program. It allows the user to discover the name, types, and properties of GUI components of the host application at runtime.
- **SharpDevelop**: an open source IDE for .NET programming.

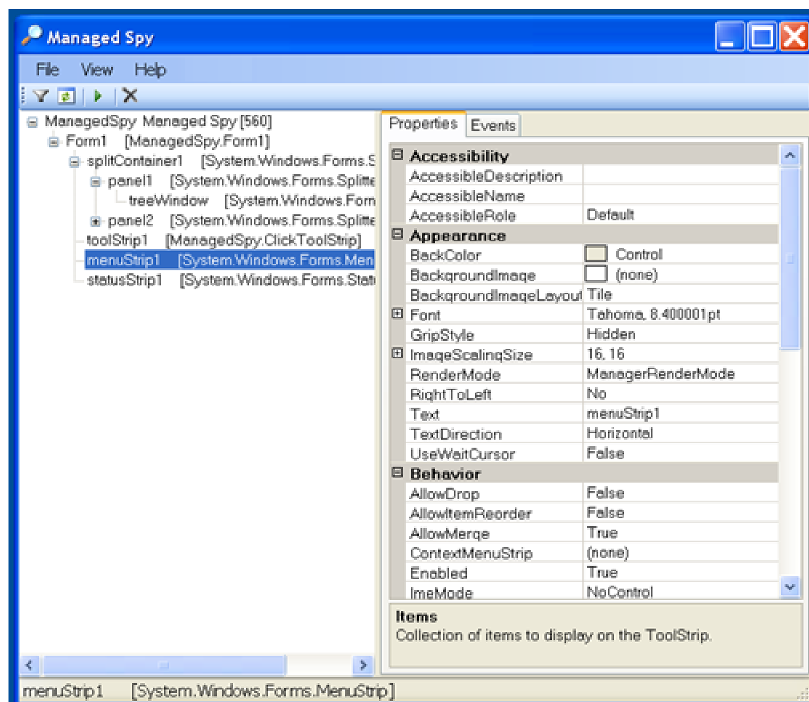


Figure 5-1 Screenshot of Managed Spy

For WADE, we provided the runtime add-on manager, and the WADE IDE with the GUI editor. Notice the difference between the two approaches mainly lies in the form of supporting tools: Scotty simulator uses tools such as ManagedSpy to retrieve the

widgets and their properties, while WADE uses the IDE to clone and GUI editor to directly edit the widgets.

5.2 Participants

Eight volunteers (indexed P1-P8; 7 males, 1 female) ranging from 21 to 32 years old ($\mu = 25.5$, $\sigma = 3.34$) participated in this study. All participants were experienced computer users and programmers. However, only 2 users had more than 3 years of GUI programming experience.

5.3 Apparatus

The experiment was conducted using a DELL Optiplex 990 Desktop computer running MS Windows XP with 4 GB RAM and Intel Core i7-2600-3.40 GHz CPU. A Dell E2211H monitor and a USB optical mouse and a standard keyboard were used as the input/output devices. Our software was implemented in C# using Microsoft Visual Studio.

5.4 Experimental protocol

Each participant was given a tutorial demonstration and three practice tasks similar to the experimental tasks in order to familiarize him/herself using the Scotty simulator and WADE before the actual experiment started. For each approach, we provided a manual with the necessary information for the users to complete the tasks. The manual for the Scotty-like approach included step-by-step instructions for (i) accessing the GUI window and child widgets,(ii) changing widget properties using the information retrieved by ManagedSpy, (iii) hiding items, (iv) adding new widgets, and (v) using the add-on manager to insert DLLs back to the host application. The WADE manual included instructions on how to (i) trigger commands to inject the add-on manager DLL, (ii) clone the host application, (iii) write GUI modifications to a DLL and (iv) load this DLL back to the host program.

Note that the instruction we provided made code-based modifications (as with the Scotty simulator) much easier, since in real world scenarios, the method to achieve GUI modifications is not obvious and needs to be figured out in a trial and error fashion. However, to facilitate novice GUI programmers to complete the tasks, we provided all the requisite information in the user manual.

A description of the tasks to be completed using (a) our Scotty simulator and (b) WADE in the actual experiment were as follows:

- **Personalized reconfiguration**: In the first task, users were required to rename two menu items, hide three menu items, change the font size and style of the main menu bar, and change the representational picture for a widget.
- **Adding functionality via add-ons**: For the second task, users were required to add a new widget on the icon bar and associate the `Undo all' functionality with the widget: once the 'Undo all' widget is clicked, it will undo all user modifications for a particular session.

A within-participants design was used. Participants were randomly assigned to two groups of four participants each. Half the participants performed the two tasks with the Scotty simulator first, while the other half performed the tasks with WADE first. Each participant performed the entire experiment in one sitting, with optional breaks between tasks, in approximately 1-2 hours. In summary, the design was as follows (excluding practices):8 subjects * 2 programming approaches (Scotty-like vs. WADE) * 2 tasks (GUI reconfiguration, add-on development) = 32 tasks in total.

Dependent variables were time spent on the tasks, whether the task is successful or not, and participants' subjective preferences in their post-experiment questionnaire. Time spent on the task is measured as the time duration between task instruction and task completion/forfeiture. A task is considered as completed if task instructions are

executed successfully. The questions listed in the post-experiment questionnaire required users to provide Likert scale-based (a scale of 1-5 was used) ratings concerning various aspects of the tasks (e.g., Approach A was more intuitive/easier to do than Approach B) and open ended questions concerning user preferences, usability and users' opinion on the relative strengths and weaknesses of both approaches.

5.5 Quantitative Measures

We compared the results in terms of (a) accuracy and (b) time spent on each task.

Accuracy: Seven participants finished all tasks, while one participant only finished the first task using both approaches. Therefore, there is no difference between the two approaches in terms of accuracy. However, the number of attempts was logged. On an average, participants spent 1.14 attempts to complete a task using WADE while they took 1.72 attempts with the Scotty-like approach. This indicated that it's easier to make mistakes using the Scotty-like approach than WADE.

Time to task completion: We conducted a 2x2 repeated measures ANOVA (Girden 1991) on the task-completion times with the type of approach (Approach A/B) and task type (reconfiguration/add-on integration) as the relevant factors. We found a significant main effect of approach ($F_{1,7} = 31.41$, $p<.01$). Pairwise t-tests showed that the time to task completion using WADE (264.4 s) is significantly (about 2.4 times) faster than that of the Scotty-like approach (639 s), indicating the significant advantage of using the WADE IDE. However, no significant effects of task type or interactions between the task difficulty and approach type were observed.

Preferences: After the experiments, participants were asked to rate various aspects of the two approaches on a 5-point Likert scale. In all, they answered four questions concerning the *usefulness*, *user productivity*, *learnability* and *overall satisfaction* of the two approaches. WADE received a minimum average score of 4.75 on all counts.

On the other hand, the Scotty-like approach received a highest score of 3.25 for *usefulness*, and a lowest score of 2.25 on *user productivity*.

While the obtained results demonstrate WADE's significant advantages over Scotty-like approaches for both types of tasks, the fact that the task type has no influence on the time to task completion is surprising. One would expect that WADE will perform better with the reconfiguration task than the add-on development task, since the former is more likely to benefit from a WYSIWYG editor- perhaps the effect of the task type would have been more pronounced had more participants been included in the user study.

5.6 Qualitative Analysis

In order to determine the reasons why participants preferred and performed better in the WADE approach, procedures to observe how participants completed the tasks were carefully followed. As summarized in Figure 5.2, all participants followed a similar work flow:

- First, participants opened a provided project template, which was identical for both approaches.
- Next, the WADE approach required an additional step of cloning the host program into IDE. In this step, participants only needed to click "Start listening" in IDE and click "Clone Me" in the host program. The actual cloning took less than five seconds. Therefore, the only additional step in the WADE approach resulted in less than half a minute of overhead. In participants' feedback, no one mentioned this step was a difficulty.
- After reading descriptions of the next subtask, participants had to locate the target widget. In the task descriptions, participants were shown screenshots of target widgets and the names of these widgets. In the WADE approach, par-

ticipants could easily find the target widget and click the widget in IDE to confirm it name. However, this step was much more time consuming in the Scotty-like approach, since participants had to go through the widget tree in Managed Spy to confirm the name. This difficulty of using Scotty-like was pointed out by three participants. "*I had problems identifying objects and their children*." (P5) "*I have to manually count the menu/toolstrip items to access them in code*." (P6) "*B (Scotty-like) takes time to look for the item to modify, and you'd better understand the component structure first; A (WADE) is easier to find what you need and what to modify by looking in the list or clicking*." (P8)

- To finish the task requirements, participants had to modify the widget properties or add a new function into the host program. This step was the essential difference between WADE and Scotty-like. In WADE, participants could simply click target widgets, and browse the property editor in IDE to modify a property. However, in Scotty-like, all participants, including those with more than six months' experience of WinForms programming, had to refer to a programming manual to recall the implementation of modifying properties for each subtask. Note that the programming manual provided implementations of all properties involved in the experiment. Participants just needed to find the correct one, and replace sample widget name with the real target widget name which they found within Managed Spy. However, the process of replacing the widget name introduced considerable overhead, since participants often typed a wrong widget name. As mentioned in section 5.5, participants took 1.72 attempts in Scotty-like and 1.14 attempts in WADE. The major reason for this was that in Scotty-like, if participants entered a wrong widget name, Sharp Develop did not have the context of the whole cloned

project for code analysis, so it was not able to warn participants at compiling. The typos could only be found during run time. One participant highlighted this point, "*Can't check for routime / syntax errors.*"(P3)

- Except for the overhead caused by typos, modifying widget properties itself was significantly simplified by the graphic UI editor of WADE. All participants strongly expressed this opinion in their own words, such as "*Approach B (WADE) is direct manipulation, easy, faster, and intuitive*" (P1); "*Coding based approach (Scotty-like) is very tedious*" (P1,2,3,7); "*B (WADE) is very easy to use. As a novice, I would create plugin with my choice of features & fonts in almost no time*" (P5). One participant gave a very positive overall comment, "*Very easy & fast! Very cool! No/very little coding involved*" (P7). A little surprisingly, even for the second task, which required participants to write a new function "Undo All", participants also completed it more than two times faster in the WADE approach than in Scotty-like. The reason may be that the function was not complicated to implement, so the time saved by IDE's event handler template was an important factor.

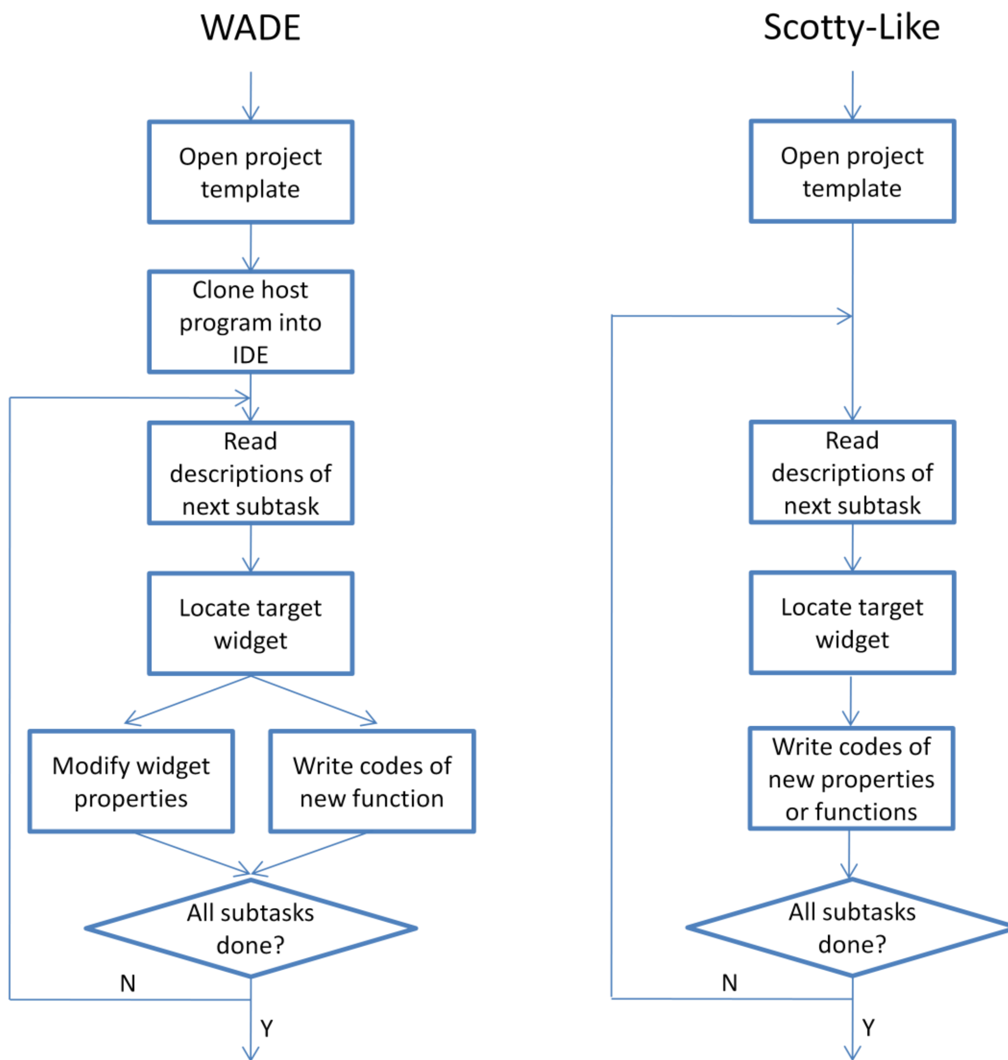


Figure 5-2 Work flow of WADE and Scotty-like approaches

According to our observations, the main advantages that WADE provides over Scotty-like are:

- Ability to directly locate target widgets
- Graphic UI editor which significantly decreases the time spent in modifying UI properties
- Event handler templates that saves the time of writing new functions
- Fully functional code analysis that can reduce typo risks

On the other hand, P1 mentioned an advantage of the Scotty-like approach: when performing the same modification to a batch of widgets, the coding bases approach can use iteration to simplify the task. For example, if users want to realize a translation for all menu items, they can use iteration to go through all menu items and call a Google Translation API to return the translated label. This is one scenario whereby people may prefer coding. Additionally, another participant felt that the graphic UI editor in WADE presents all properties of a widget at the same time, which is a little overwhelming. "*Too many options for change / to edit at the sidebar (PropertyEditor). It feels overwhelming at first.*" (P3)

There were also some suggestions from the participants for further enhancing the user experience of third party add-on development. P1 thought that merging two approaches would be a better solution since he could then use the WYSIWYG editor and iterations of coding at the same. This idea is actually used in standalone software development: developers enjoy combining the advantages of both WYSIWYG editors and coding. Two participants wished that IDE could provide more advanced support for handling events, in addition to the WYSIWYG editor. "*Hopefully there's a way to eliminate / simplify coding in the handling of events*." (P3) "*It's even better if I don't have to code the event handle method*." (P7) One participant appreciated WADE very much and believed it was worthwhile to make WADE available for all add-ons. "*Approach A (WADE) should be implemented as default option for all available plugins, so that they can be merged & compiled for better & wider use.*" (P5)

5.7 Summary

The results of the user study clearly demonstrate the advantage of WADE over Scotty-like approach for both reconfiguration and add-on development tasks. While the conclusion of the study is not surprising, as it is expected that the direct WYSIWYG

GUI editing will be easier than hard core code hacking, we are surprised to see the large difference (2.4 times performance difference) of the two approaches for experienced programmers with relatively simple modifications even if the participant has already gathered all the necessary knowledge to complete the task. In real world scenarios, the type of knowledge we provided to the participant is often not given, thus significant more time is needed to search and verify the possible feasible approaches, making a simple GUI modification a relative challenging task that very few people would like to attempt. Therefore, it shows that although a Scotty-like approach can accomplish both the reconfiguration and add-on development tasks for third party applications, the entry barrier for using Scotty-like approach is sufficiently high that only a few hardcore hackers would dare to attempt. On the other hand, WADE made it much easier to perform simple modifications and enhancements, significantly lower the entry barrier to perform third-party add-on development.

However, if the particular add-on involves a lot of background processing and only few modifications to GUI elements, then both approaches would require significant amount of time to work on the backend programming logic. The amount of time saved with easier GUI manipulation will become less significant.

In summary, to perform simple GUI reconfiguration tasks or add-on development, WADE can save an average of 60% or more time as compared with Scotty-like approach, making it a preferred and recommended approach for such tasks.

Chapter 6 Discussion

6.1 Extension to other frameworks and platforms

Given the advantages of WADE, one may be interested in extending it to other frameworks and platforms. We now discuss such feasibilities. The key components of WADE involves: 1) runtime code observation and intervention, and 2) GUI clone and modification analysis. We discuss how to realize these concepts on Windows before considering other platforms:

The details of how to achieve runtime code observation and intervention on Mac OS' Cocoa framework has been discussed in detail by [5]. We will discuss how to achieve this on windows for other frameworks. The key approach for runtime code observation and intervention on Windows is DLL injection, which can be achieved in a number of ways. Some of the methods (such as hijacking an existing thread to execute the injected code at debugging time) are either inconvenient or infeasible for the purpose of modifying third party applications, thus we only discuss the two practical methods below.

1) Registry key-based injection by listing the DLL under a specific registry key to be loaded to the application process. In Windows NT, 2000, and XP, this can be achieved by list the DLL under the register key (Working with the AppInit_DLLs registry value n.d.):

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs

In Windows Vista and Windows 7, this feature is disabled by default, but can be achieved through code signing.

2) System hook-based injection by using methods such as SetWindowsHookEx, which can be used by all versions of Windows.

These two methods operate at different levels of the application and offer different tradeoffs: the registry key-based approach injects the DLL at the process level, while the System hook-based approach penetrates to the thread level. To understand the tradeoffs between the two methods, we first need to explain the difference between processes and threads.

Processes are independent execution units that contain their own state information, use their own address spaces, and only interact with each other via interprocess communication mechanisms (generally managed by the operating system). Processes are an architectural construct. Most applications today typically only have one process. In contrast, a thread is a coding construct that doesn't affect the architecture of an application. A single process might contain multiple threads; all threads within a process share the same state and same memory space and can communicate with each other directly, because they share the same variables. However, as we have previously explained, modification of the UI thread can be risky and needs to be managed with care. Therefore, some frameworks do not allow modification of elements in the UI thread by other threads. Due to this reason, DLL injected to the process level may not be able to modify the UI elements depending on the framework used.

On the other hand, in order to inject DLL to a thread, one first needs to know if its hosting process is currently running or not. Without this information, it is impossible to inject the DLL successfully. In order to obtain this information, the developer needs to create an additional background monitoring process, which can be tedious

and consumes additional system resources. The registry key based approach relies on the native windows' support to automatically injecting the DLL to the process, so no additional efforts are needed

In summary, the trade-offs between the two approaches are: registry key based approach is simpler to implement, but can be restrictive in terms of accessing the UI elements while the system hook based approach can directly access the UI thread, but require additional efforts to manage. Using these two methods, theoretically speaking, any frameworks on Windows can adopt the same approach as WADE to modify a third party application's appearance or behavior at runtime. To implement the runtime code observation and intervention support for other frameworks, one first needs to find out whether or not that framework supports cross thread modification of UI elements and decide on which DLL injection method to implement.

Once that is possible, the second part of our approach involves creating an enhanced IDE with GUI editor that can talk to the injected DLL and clone the GUI hierarchy of the host application. This is a framework dependent process. Specific implementation needs to be done to make it work for different frameworks. For example, if a developer wants to realize the WADE functionality for the QT framework, it needs to find an IDE that supports GUI editing for the QT framework, write an add-on to so that it can import the GUI hierarchy from the third library application, analyze its changes, and compile into a QT specific DLL to be loaded back to the host application. This step will become easier if the particular framework has working IDEs supports add-on development and already has GUI editor support. However, if there are no such IDEs available, it will be a tremendous effort to develop an IDE with such capabilities from scratch.

For Unix and Mac systems:

- On Unix-like operating systems with the dynamic linker based on ld.so (on BSD) and ld-linux.so (on Linux), arbitrary libraries can be linked to a new process by giving the library's pathname in the LD PRELOAD environment variable, which can be set globally or individually for a single process.
- For Cocoa applications running on Mac OS X, Input Managers enable applications to change the way that Cocoa handles user input. Through this mechanism, another process can gain access to the underlying Objective-C runtime of the host applications, such as (Eagan, Beaudouin-Lafon and Mackay 2011)

Once DLLs are successfully injected, the same approach described above can be used to create the add-on architecture and IDE with GUI editor support, although the implementation details will differ depending on the platform.

Chapter 7 Conclusion

7.1 Contribution

To summarize, the goals of this thesis were to:

- Provide a *holistic solution* (through a scaffolded approach to program modification using DLL-injection + WYSIWYG-style IDE-based editing) that significantly lowers the knowledge and effort required to modify third-party applications and therefore, considerably enhances usability of such tools as confirmed by the user-study. The primary and unique contribution of WADE is not its components or the techniques employed, but the fact that WADE represents a holistic solution for runtime program modification that requires (i) designing a suitable architecture (IDE), (ii) developing individual components (DLL injection, Injected and IDE add-on managers, etc.), and (iii) integrating them.
- Develop a variety of utility add-ons that can benefit users and developers, including the Property Editor add-on that allows appearance modifications to third-party UIs without the need to write a single line of code so that novice users can also use runtime modification according to personal preferences.
- Demonstrate the effectiveness of the WADE IDE for modifying existing applications through a user-study.
- Represent the first solution for runtime modification on the Windows operating system.

7.2 Limitations

While the WADE GUI editor has shown a lot of promise in simplifying the reconfiguration and add-on development process, it nevertheless has a number of limitations.

7.2.1 Custom Widget Clone and Modification

First, the current WADE implementation only supports the cloning and modification of standard widgets. If the third party software contains a custom widget, the cloning may not always work. This is because custom widgets often have derived custom properties and behaviors that are not recognized by the GUI editor; they therefore cannot be properly displayed in the GUI editor, making direct editing difficult. However, not all custom widgets are unrecognizable. If the basic properties of the custom widget are derived from a standard widget, the GUI editor can still display it and allow modification of those properties.

For example, if a custom widget MyGroupBox is derived from the standard widget GroupBox, it inherits its parent's Size, Location, Label Text, and other properties, all of which can be recognized by the GUI editor. However, MyGroupBox may choose to have a custom paint method that gives it a different look. This feature is not recognizable by the GUI editor and cannot be changed in a WYSIWYG fashion in our current implementation.

Certain properties of custom widgets can be recognized by the GUI editor, but user modification of these properties using the GUI editor cannot be easily applied back to the host application. For example, the Paint.NET's ToolStrip widget is a custom widget derived from the standard ToolStrip widget. It contains code to reset its location property to the left most position in the window. If a developer changs its location in the GUI editor, that change cannot be reflected in the host application.

7.2.2 Dynamic Widgets

Dynamic widgets are advanced features introduced in modern frameworks. Instead of fixing their content and style at compile time, dynamic widgets determine them at runtime based on an external data source. Dynamic widgets can save time and effort if developers want to create several widgets in a similar style but with different content. However, since the content of widget is determined at runtime, modification of the content in the GUI editor is applied back to the original application.

7.3 Future Work

The WADE IDE presented in this thesis is useful for realizing a variety of GUI-based modifications in existing software, including GUI reconfiguration and synthesis of a number of utility add-ons. Also, the presented user study confirms that while these modifications can be achieved using alternative approaches, WADE significantly lowers the requisite knowledge and effort barrier. Future work involves extending the current implementation to other OS platforms and widening WADE support to handle custom and dynamic widgets.

Bibliography

Baudisch, P., et al. "Phosphor: Explaining Transitions in the User Interface using Afterglow Effects." *UIST*. 2006. 169-178.

Begole, J.M.A. "Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems." PhD thesis, Virginia Polytechnic Institute and State University, 1998.

Berdajs, J., and Z. Bosnic. "Extending applications using an advanced approach to DLL injection and API hooking." *Software: Practice and Experience* 40, no. 7 (2010): 567–584.

Besacier, G., and F. Vernier. "Toward user interface virtualization: legacy applications and innovative interaction systems." *EICS*. 2009. 157–166.

Birrell, A.D., and B.J. Nelson. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems* 2, no. 1 (1984): 39-59.

Bray, T., J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. "Extensible markup language (XML)." *World Wide Web Journal* 2, no. 4 (1997): 27-66.

Chapuis, O., and N. Roussel. "Metisse is not 3D desktop!" *UIST*. 2005. 13–22.

CrankHank. *DLL Injection and function interception tutorial.* http://www.codeproject.com/Articles/5178/DLL-Injection-and-function-interception-tutorial (accessed 11 10, 2012).

Dietrich, D.C. *Writing effect plug-ins for Paint.NET 2.1 in C#.* http://www.codeproject.com/Articles/9200/Writing-effect-plug-ins-for-Paint-NET-2-1-in-C (accessed 12 3, 2012).

Dixon, M., and J. Fogarty. "Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure." *CHI.* 2010. 1525–1534.

Eagan, J.R., M. Beaudouin-Lafon, and W.E. Mackay. "Cracking the cocoa nut: user interface programming at runtime." *UIST.* 2011. 225–234.

Edwards, W.K., E.D. Mynatt, and K. Stockton. "Providing access to graphical user interfaces —— not graphical screens." *Assets.* 1994. 47-54.

Edwards, W.K., S.E. Hudson, J. Marinacci, R. Rodenstein, T. Rodriguez, and I. Smith. "Systematic output modification in a 2d user interface toolkit." *UIST.* 1997. 151–158.

Georgescu, M., and D. Milodin. "Techniques of Improving Open Source Software Tools." *Open Source Science Journal* 2, no. 3 (2010): 34-48.

Girden, E.R. *ANOVA: Repeated measures.* Sage Publications, Incorporated, 1991.

Grossman, T., and R. Balakrishnan. "The Bubble Cursor: Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area." *CHI.* 2005. 281-290.

Heege, M. "Assembly Startup and Runtime Initialization." In *Expert C++/CLI*, 279-302. Springer, 2007.

Holm, C., M. Kruger, and B. Spuida. *Dissecting a C# Application Inside SharpDevelop.* Apress L.P., 2004.

How To Inject a Managed .NET Assembly (DLL) Into Another Process. http://www.codingthewheel.com/archives/how-to-inject-a-managed-assembly-dll (accessed 11 10, 2012).

Hutchings, D.G., and J. Stasko. *New operations for display space management and window management.* Georgia Institute of Technology Technical Report GIT-GVU-02-18, 2002.

Hutchins, E.L., J.D. Hollan, and D.A. Norman. "Direct manipulation interfaces." *Human-computer interaction* 1, no. 4 (1985): 311-338.

Interprocess Communications (Windows). http://msdn.microsoft.com/en-us/library/windows/desktop/aa365574%28v=vs.85%29.aspx (accessed 11 10, 2012).

Jahnke, J., and A. Zundorf. *Rewriting poor design patterns by good design patterns.* ESEC/FSE'97 Workshop on Object-Oriented Reengineering, 1997.

Kandogan, E., and B. Schneiderman. "Elastic windows: Evaluation of multi-window operations." *CHI.* 1997. 250-257.

Kuster, R. *Three Ways to Inject Your Code into Another Process.* http://www.codeproject.com/Articles/4610/Three-Ways-to-Inject-Your-Code-into-Another-Proces (accessed 11 10, 2012).

Lie, H.W., and B. Bos. *Cascading style sheets.* Addison-Wesley, 1997.

Mackay, W.E. "Triggers and barriers to customizing software." *CHI.* 1991. 153–160.

Media Interaction Lab, University of Applied Sciences Upper Austria. *New Open Source Eye Tracking Software by Media Interaction Lab .* http://www.eyetechds.com/new-eye-tracking-software-by-at-media-lab (accessed 12 3, 2012).

Miah, T., and J.L. Alty. "Vanishing Windows — a technique for adaptive." *Interacting with Computers* 12 (2000): 337–355.

Myers, B.A., and M.B. Rosson. "Survey on user interface programming." *SIGCHI conference on Human factors in computing systems.* 1992. 195-202.

Myers, B.A., and W. Buxton. "Creating highly-interactive and graphical user interfaces by demonstration." *ACM SIGGRAPH Computer Graphics* 20, no. 4 (1986): 249-258.

Network, Microsoft Developer. *.NET Framework Conceptual Overview* . http://msdn.microsoft.com/en-US/library/zw4w595w%28v=vs.80%29.aspx (accessed 11 10, 2012).

Newcomer, J.M. *Hooks and DLLs.* http://www.codeproject.com/Articles/1037/Hooks-and-DLLs (accessed 11 10, 2012).

Pietrek, M. "Learn system-level Win32 coding techniques by writing an APIs py program." *Microsoft System Journal* 9 (1994): 1-22.

Postel, J. *User datagram protocol.* Information Sciences Institute, 1980.

Pulley, R. *Extending task manager with DLL injection.* http://www.codeproject.com/Articles/10437/Extending-Task-Manager-with-DLL-Injection (accessed 11 10, 2012).

Richardson, T., Q. Stafford-Fraser, K.R. Wood, and A. Hopper. "Virtual network computing." *IEEE Internet Computing*, Jan/Feb 1998: 33–38.

Richter, J. "Load your 32-bit DLL into another process's address space using INJLIB." *Microsoft Systems Journal* 9, no. 5 (1994): 1-10.

Robinson, M. "Design for unanticipated use." *ECSCW.* 1993. 187–202.

Russell, R., D. Quinlan, and C. Yeoh. *Filesystem Hierarchy Standard.* Filesystem Hierarchy Standard Group, 2004.

Shneiderman, B. "direct manipulation: a step beyond programming languages." In *Sparks of Innovation in Human-Computer Interaction*. Ablex Publ., 1993.

Silver, A.H. *WordPress 2. 7 Complete.* Packt Publishing Ltd., 2009.

Stevens, W.R., and G.R. Wright. *TCP/IP Illustrated: the protocols.* Addison-Wesley Professional, 1994.

Stuerzlinger, W., O. Chapuis, D. Phillips, and N. Roussel. "User Interface Facades: Towards Fully Adaptable User Interfaces." *UIST.* 2006. 309–318.

Tan, D.S., B. Meyers, and M. Czerwinski. "Wincuts: manipulating arbitrary window regions for more effective use of screen space." *CHI.* 2004. 1525–1528.

Wallach, R.S. "Gemini Lite: a non-intrusive debugger for Windows NT." *4th USENIX Windows Systems Symposium.* 2000.

Wikipedia. *Heat map.* http://en.wikipedia.org/wiki/Heat_map (accessed 12 3, 2012).

Wilkinson, L., and M. Friendly. "The history of the cluster heat map." *The American Statistician* 63, no. 2 (2009): 179-184.

Working with the AppInit_DLLs registry value. http://support.microsoft.com/kb/197571 (accessed 11 10, 2012).

Wu, L. "Implementation of Self-Defined JSP Plug-in Based on Notepad++." *Computer Programming Skills & Maintenance* 6 (2010).

Yeh, T., T.-H. Chang, and R.C. Miller. "Sikuli: using gui screenshots for search and automation." *UIST*. 2009. 183–192.

Zhao, S., and R. Balakrishnan. "Simple vs. compound mark hierarchical marking menus." *UIST*. 2004. 33-42.

Zhao, S., M. Agrawala, and K. Kinckley. "Zone and polygon menus: using relative position to increase the breadth of multi-stroke marking menus." *CHI*. 2006. 1077-1086.