WADE: Simplified GUI Add-on Development for Third-party Software

Xiaojun Meng[1], Shengdong Zhao[1], Yongfeng Huang[1], Zhongyuan Zhang[1], James R. Eagan[2], Ramanathan Subramanian[3]

[1]NUS-HCI Lab, National University of Singapore, [2]Télécom ParisTech & CNRS LTCI UMR 5141, [3]ADSC, University of Illinois at Urbana-Champaign

[1]{xiaojun,zhaosd}@comp.nus.edu.sg,[1]xgjonathan@gmail.com,[1]zhang_zhongyuan@nus.edu.sg

[2]james.eagan@telecom-paristech.fr,[3]subramanian.r@adsc.com.sg

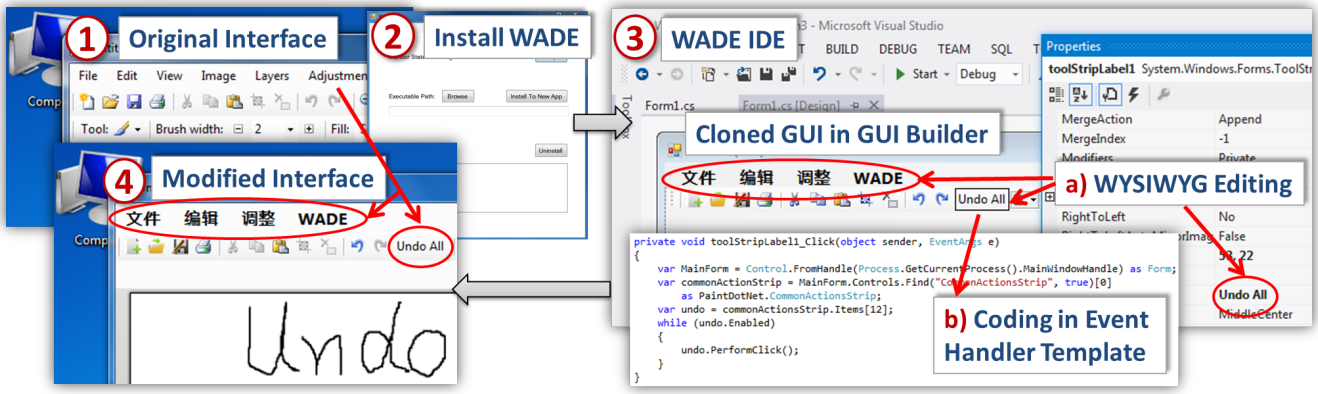

Figure 1: WADE overview: 1) Kevin wants to make the following modifications to the original Paint.NET interface i) change menu labels from English to Chinese, ii) remove unused menus and icons, and iii) add a new "undo all" function. To this end, he 2) Installs WADE in Paint.NET, 3) Clones the Paint.NET GUI into the GUI builder of a WADE-supported IDE. Then, he (a) modifies the GUI using a WYSIWYG editor, and (b) writes code associated with the "undo all" widget via the event handler template. 4) All the above changes are compiled to an add-on that can be installed into Paint.NET for easy and convenient future use.

ABSTRACT

We present the WADE Integrated Development Environment (IDE), which simplifies the modification of the interface and functionality of existing third-party software without access to source code. WADE clones the Graphical User Interface (GUI) of a host program through dynamic-link library (DLL) injection in order to enable (1) WYSIWYG modification of the GUI and (2) the modification of software functionality. We compare WADE with an alternative state-of-the-art runtime toolkit overloading approach in a user-study, finding that WADE significantly simplifies the task of GUI-based add-on development.

Author Keywords

WADE; GUI; Add-on Integration; WYSIWYG; IDE

ACM Classification Keywords

H.5.2. Information Interfaces and Presentation: User Interfaces

General Terms

Human Factors

INTRODUCTION

Software rarely fulfills the needs of all users all the time [7, 12]. 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 (*e.g.*, via preferences panes or dot files), or
- Software architecture for incorporating add-ons (*e.g.*, using plugins, scripts and/or extensions) to enhance/modify the behavior of the original application.

While these approaches can provide users with a great deal of control, every approach necessitates additional effort from the software developers to explicitly provide customization support at the software development stage. For example, plugins, scripting interfaces and extensions require the developer to provide and maintain an external API to their software, which may potentially require

CHI 2014, April 26–May 1, 2014, Toronto, ON, Canada.

http://dx.doi.org/10.1145/2556288.2557349

maintaining an additional and 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 [1]. To address this limitation, 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 to *an external API*. These approaches typically work by either: 1) operating on the surface-level of the interface, intercepting input events and output pixels before they are delivered to the application (*e.g.*, Prefab [2, 3], Façade [14]), or 2) integrating with the toolkit to gain access to the internal program structures (*e.g.*, Scotty [4], SubArctic [5]). For convenience, we call the former as *surface-based approaches* and the latter as *toolkit-based deep approaches*.

Surface-based approaches allow modifications to GUI elements without access to the internal structure of an application. For example, Façade allows for reconfiguring GUI elements via a simple drag and drop interface [14]. However, such approaches are limited by their ability to infer the structure and functionality of the interface because they do not have access to the internal program objects or their semantics. *E.g.*, adding new functionality or modifying the behavior of a GUI widget is difficult to accomplish using surface-based approaches [4].

This limitation can be overcome to some extent by toolkit-based deep approaches such as Scotty [4] or SubArctic [5], which operate below the surface of the program to reveal the underlying program logic and functionalities. This allows them to alter the system's appearance and behaviors beyond the surface level. However, toolkit-based deep approaches can be challenging to use. They require a thorough understanding of the relevant parts of the system in order to realize the desired behavior. Even for experienced developers, much effort is needed to make relatively simple modifications to third party software.

Therefore, there exists a trade-off between generalizability, ease of use, and power (the ability to perform deeper modifications). While all previous approaches have their advantages, additional solutions are still needed to better balance the power and ease of use for runtime modification of third-party software.

In this paper, we propose WADE, a simplified and WSYWYG Add-on Development Environment that can ease the task of modifying GUI-based functions in existing software with or without source code, while still enabling developers to make deep changes to the software behavior. To achieve this, WADE injects a dynamically-linked library (DLL) into the host program to retrieve the GUI hierarchy of the host program. It then clones the interface in the IDE so that properties of GUI elements can be directly modified. Furthermore, WADE provides scaffolding to directly associate event handlers to existing widgets, so that enhancing/modifying software behavior becomes simpler.

Figure 1 shows an example add-on development scenario using WADE. Currently, WADE supports add-on development using both the open source SharpDevelop 4.2 and the Microsoft Visual Studio 2012 Ultimate IDEs for *Windows Form* applications on the Windows XP and Windows 7 platforms.

We conducted an experiment to compare WADE with a Scotty-like toolkit-based deep approach for modifying third party applications. Our results show that users subjectively found WADE much easier to use, and were objectively able to develop GUI-based modifications 2.4 times faster than the alternative approach on average. To summarize, the contributions of this work are:

- We present the WADE prototype along with its software architecture as an integrated solution for significantly facilitating add-on creation for third party software without source code.
- The WADE IDE provides scaffolding for code-based GUI modification through template generation, thereby enabling robust implementation of the complex boilerplate associated with runtime modification.
- We present the results of an empirical comparison between WADE and the state-of-the-art Scotty approach for modifying software [4], which shows that WADE is significantly faster for GUI modifications.

RELATED WORK

As previously mentioned, surface-based adaptation [2, 3, 14, 15] and toolkit-based subsurface modification [4, 5, 16] are the two main approaches that support third-party application modifications without access to the software's source code. As a comprehensive review of the different variants of these two approaches has already been discussed in Eagan *et al*. [4], we now highlight those works most relevant to WADE.

Surface-level modification

Surface-level modifications do not require any support by the application developer. Instead, they operate on the interface that is presented to the user and the input events he or she provides. For example, Yeh *et al*.'s Sikuli scripting environment [1] allows users to write scripts that reference screenshots of particular controls, to refer to existing application elements.

Stuerzlinger *et al*.'s UI Façades [14] intercept individual widgets as they interact with the window server. This allows a developer to easily replace them at the window server level with an alternate implementation, such as by regrouping together widgets from different applications or replacing a radio button with a pop-down menu. Dixon and Fogarty's Prefab [2] 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, Prefab can enable alternate software functionality.

However, all of these solutions are limited by their ability to infer the structure and functionality of the interface. They do not have direct access to internal program objects or their semantics. As a result, it is typically challenging for such approaches to make modifications that alter both GUI elements and their underlying program logic. Such limitations can be overcome to some extent by toolkit-based deep modification approaches.

Toolkit-based deep modification

Edwards *et al.*'s SubArctic toolkit [5] extends Java's AWT to provide explicit hooks that allow third-party developers to add new UI modifications. These hooks provide specific support for extensibility, allowing a third-party developer to add new functionality to existing applications built with the SubArctic toolkit, without explicit software support. However, UI modifications are only feasible for applications built using the SubArctic toolkit. For other types of applications, such modifications become infeasible.

Eagan *et al.*'s Scotty [4] uses injection to perform runtime toolkit overloading, in which an existing toolkit is altered specifically to provide explicit support for modifications. It provides a *meta*-toolkit for developers to modify existing third-party applications. Third-party developers must, however, explicitly *inspect* and *make sense of* the existing application before eventually applying acquired knowledge in a separate coding environment [6, 9]. This process can be complex, creating barriers that limit such modifications to experienced and dedicated programmers.

USING WADE

Our goal with WADE was to create an interface that unifies the various tasks and tools involved in creating third-party program modifications. In contrast to Scotty, where *sense-making* and *coding* are independent, WADE integrates the two phases into a single environment, making software modification more user-friendly and practicable even to novice programmers.

A third-party developer can use WADE to make a variety of modifications to an application, such as a) basic property changes to a GUI's widgets, b) altering actions associated with interface elements, and c) adding entirely new functionalities. We demonstrate the utility of WADE through the following scenarios. All scenarios have been implemented using WADE.

Language localization and template creation

Kevin has created a diary template for Notepad.NET and wants to share it with his Russian friend Ivanov, who is not comfortable with English. Unfortunately, Notepad.NET does not currently have a Russian translation, so Kevin loads the WADE property editor add-on into Notepad.NET. In the property editor, he systematically changes each widget's label to its Russian translation, as shown in Figure 2. He then exports those changes to a new add-on component that Ivanov can load into his English copy.

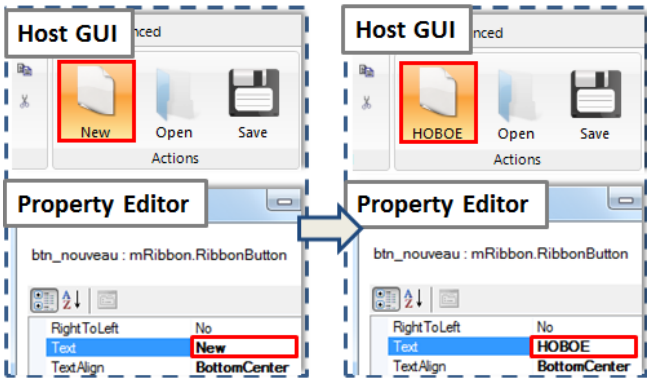

Figure 2: Changing the label of the program from English to Russian using WADE's property editor. The user first selects the GUI widget (the *New* button), and types its Russian name in the text field of the property editor. Changes to the text are immediately reflected in the host GUI.

Kevin then decides to add a toolbar button as a shortcut to the new diary template he has created. While the property editor can alter properties of existing widgets, it cannot add new widgets. Kevin loads the WADE IDE and chooses the **Clone GUI** command to clone the Notepad.NET interface into a new project. Using the WADE add-on that Kevin already loaded into Notepad.NET for translation, the clone command walks the interface hierarchy and serializes it to the WADE IDE. In the WADE IDE's GUI builder, Kevin then adds a new toolbar button for the diary. He then uses WADE to generate an event handler template, into which he writes the code to load his new diary template.

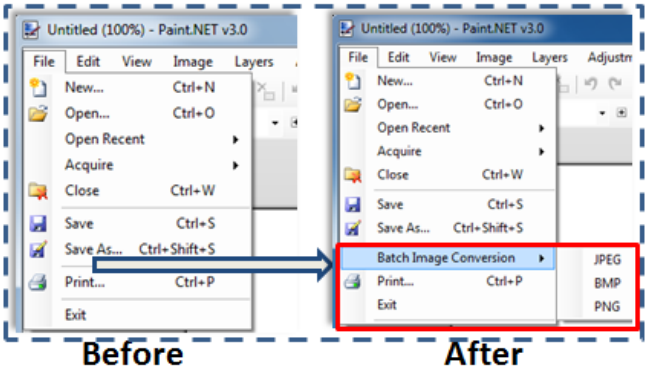

Figure 3: Using WADE, one can easily add an external service to the host program. The utility enabling batch image conversion to JPEG, BMP and PNG is added to Paint.NET by linking to the ImageMagick graphic library.

Link to external function call

Lee took a lot of photos in RAW format on her recent trip to Toronto and wants to convert them to JPEGs so that she can open them in Paint.NET. Unfortunately, Paint.NET does not have a batch conversion interface. There is a command-line tool that offers that capability, but she can never remember the right incantation to make it work. She

clones Paint.NET's interface into WADE and adds a new **Batch Conversion** menu (Figure 3). She then uses WADE's event handler template to invoke the appropriate actions using the command line library, compiles the add-on and installs it into Paint.NET.

Discussion

The above scenarios illustrate some of the different kinds of third-party program modifications that WADE supports. In the first example, Kevin is able to provide a translation for a third-party interface for his friend Ivanov, just by using WADE's property editor add-on for existing programs. For more complex modifications to the interface, such as when Lee adds batch conversion support to Paint.NET, it is necessary to write some code for the new functionality. Here, WADE provides a) support to clone the existing interface into a new project and b) scaffolding to help Lee write her event handlers. The only code she needs to write is the code specifically related to her functionality, which she can then integrate into the cloned GUI hierarchy using WADE's GUI editor. We present the detailed implementation in the following sections.

Comparison with previous approaches

Other tools provide similar kinds of third-party program modification. Façade [14] enables the user to easily simplify an interface by removing and regrouping widgets. However, it does not support changing labels, font styles, background images, *etc*.

Prefab uses a localization example similar to Kevin's scenario in order to show the power of pixel-based approaches. However, Prefab can only access pixels but not the text, and must therefore apply a character recognition process to extract associated text strings. In contrast, WADE retrieves the original label text directly from the host application's internal structure.

The remaining modifications require deep access to the program's internal structures. As such, surface approaches such as Façade and Prefab cannot pierce through the surface to decipher these structures.

Toolkit-based subsurface approaches, such as Scotty, can accomplish all the tasks above, but do not provide the scaffolding and support of an IDE that WADE does. In order to perform language localization, for example, a developer must inspect the UI hierarchy and associate program objects to identify widgets and corresponding labels, before writing the appropriate code from scratch to change the labels to another language. WADE, on the other hand, simplifies this process by presenting a unified environment and scaffolding for many of these changes. We now describe how WADE facilitates software modifications using the GUI builder.

WADE IDE FOR ADD-ON DEVELOPMENT

While the details of developing add-ons for third-party software without source code can be complex, the basic idea involves two important aspects. First, third-party applications may not come with a pre-designed add-on architecture. Therefore, an environment should be designed in which the host application can manage and communicate with add-ons created and integrated with it at a later time. Second, because the application source code is not available, the IDE must facilitate understanding of the host application's internal structure and provide tools to support the creation of add-ons.

Injecting WADE add-on manager to host application

To achieve the first goal, WADE adopts an approach similar to Scotty's, by injecting an add-on manager into the host application's process space. While Scotty is designed to work on the Mac OS X Cocoa platform, WADE is developed for *Windows Form* applications on the Windows operating system. WADE uses the registry key binding technique to insert compiled code, in the form of a Dynamic Linked Library (DLLs), into the host application at runtime. Once loaded, the injected DLL can use the *CreateRemoteThread* method to create threads that run in the virtual address space of the host processor. This allows it to serve as an add-on manager to load and register any compiled add-ons (also in the form of DLLs) within the host application [13].

Supporting third-party add-on development

However, simply enabling external add-ons to be integrated with the host application is not enough. In order to create meaningful add-ons, a third-party programmer must make sense of an existing application, and apply that knowledge to the development of any new functionality.

Scotty provides various tools including a hierarchy browser, an object inspector, a widget picker, and an interactive interpreter (Python) to support sense-making in the *Cocoa* environment [4]. While none of the individual tools may be too difficult to use, they only provide partial answers. Knowing how and where to get the different pieces of information, and discovering how to combine them effectively to obtain a high level picture, are both tedious and challenging. Therefore, typically, only experienced programmers are able to use Scotty-like approaches.

In order to reduce the knowledge barrier involved in integrating the different tasks mentioned above, we introduce an IDE specifically for third-party add-on development. An IDE is a software application that provides comprehensive facilities to computer programmers for software development. It is designed to maximize programmer productivity by providing tightly-knit components for authoring, modifying, compiling, deploying and debugging software with similar user interfaces. The IDE, therefore, is more user-friendly and powerful as compared to multiple distinct tools provided by Scotty.

Modern IDEs often come with an integrated GUI builder (also known as GUI editor), which simplifies GUI creation by allowing the designer to arrange widgets using a drag-and-drop WYSIWYG editor. As today's user interfaces are

commonly programmed using an event-driven architecture, GUI builders also simplify creation of event-driven code, by supporting code that connects widgets with the incoming (input) and outgoing (drawing) events that trigger functions providing the application logic.

Integrating WADE with the IDE and GUI builder

However, integrating an IDE with a GUI builder into the third party add-on development process is no simple task. GUI builders in existing IDEs are designed to facilitate the creation of new interfaces from scratch, rather than to modify existing interfaces. In addition, existing GUI builders tend to assume that source code associated with the GUI components will be available. In our case, however, that crucial piece of information is missing.

In order to enable the WADE GUI builder to modify GUI components and their associated program logic for a third-party application, the following steps are needed:

1) Extract the GUI hierarchy information from the host application.
2) Send this information to the GUI builder inside of an IDE.
3) In the IDE, convert this information into a format that can be displayed as GUI widgets in the GUI builder, so programmers can manipulate them in a WYSIWYG fashion.
4) Analyze and compile the changes made by the programmer into an add-on that can correctly modify the appearance and behavior of the host application at runtime.

Before elaborating on the implementation process, we will first define a few terms.

GUI frameworks typically organize widgets into a tree. The root tree has sub-trees that represent windows and their associated widgets.

We term the root tree of the host application as *host GUI hierarchy*, which contains many *host widget sub-trees*. Each *host widget sub-tree* represents a window that has a hierarchy of *host widgets*.

We replicate the *host GUI hierarchy* inside the IDE's GUI builder. The replicated copy is called the *cloned GUI hierarchy*, which consists of many *cloned widget sub-trees*. Each *cloned widget sub-tree* has many c*loned widgets*.

The relationship between these terms is illustrated in the left and right panels of Figure 4. We now describe in more detail the steps involved in using WADE to modify GUI components and associated program logic for a third-party applications.

Step 1: Extract GUI hierarchy information

We overload the *Injected Add-on Manager* to perform several additional steps beyond basic add-on management. In order to gain access to all of the widgets in the host GUI hierarchy, the Injected Add-on Manager walks each of these trees to extract its structure and to identify the properties (*e.g.*, name, size, location, label, etc.) of each widget in the hierarchy. We use 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.

In addition, the *Injected Add-on Manager* constructs a *component dictionary* of all the widgets of the unmodified *host GUI hierarchy* by using the *name* and *address* of each widget as a (key, value) pair in the dictionary. This information is saved as a reference point so that any potential changes made by a third-party programmer using the IDE can later be discovered.

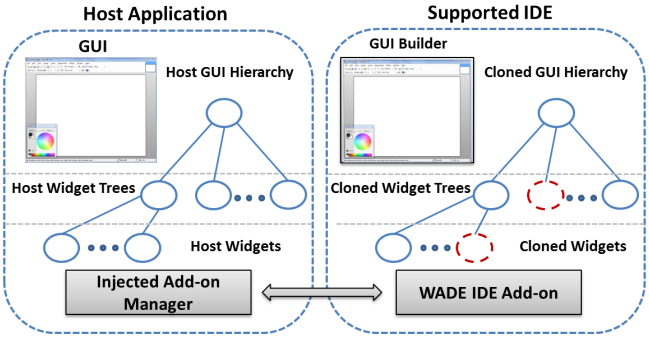

Figure 4: WADE components: the *Injected Add-on Manager* (left panel) inside the host application manages add-ons and communicates the GUI information with a compatible IDE via the *WADE IDE Add-on* component (right panel). The *WADE IDE Add-on* then clones the host application's GUI in the IDE's GUI Builder to allow WYSWYG modification of the original UI. The changes made in IDE can then be compiled into a third-party add-on to alter the appearance and behavior of the host application.

Step 2: Send information to the GUI builder

The *Injected Add-on Manager* then serializes the extracted properties of each *host widget* via the *WADE IDE Add-on* to the IDE. For most widgets, information such as name, size, location, text, *etc.* are directly sent through a basic text stream. For widgets with background images or complex structures, such information is first saved as cache files in image or XML format before being transferred over.

Step 3: Convert and present GUI information in GUI builder

After receiving complete GUI information from the *Injected Add-on Manager*, the *WADE IDE Add-on* then constructs a project with the same UI properties as extracted from the original program. With the extracted UI information, the *WADE IDE Add-on* clones the existing interface into a new project in the IDE. In our current WADE implementation, we have integrated the *WADE IDE Add-on* with SharpDevelop 4.2 and Microsoft Visual Studio 2012 Ultimate to provide code and GUI builder support. The *WADE IDE Add-on* uses the serialized information to

replicate the *host GUI hierarchy* on the canvas of the supported IDE's GUI builder.

Step 4: Analyze and apply changes

Third-party add-on developers can then modify the *cloned GUI hierarchy* in a WYSIWYG fashion. This modified *cloned GUI hierarchy* and its associated program behavior is compiled into an add-on (in a DLL) that can be loaded into the host application by the *Injected Add-on Manager*.

Using the earlier constructed *component dictionary*, the *Injected Add-on Manager* can then examine the modified *cloned GUI hierarchy* inside the add-on and apply the changes to the *host GUI hierarchy* as described by the following simplified algorithm:

```
1) make all widgets in the host GUI hierarchy invisible
2) for each cloned widget tree in the cloned GUI hierarchy:
3)     perform a breadth-first walk through all the cloned
       widgets, and for each cloned widget:
4)         try to find its corresponding host widget by looking
           up in the component dictionary using the widget
           name as the key
5)         if a corresponding host widget is found:
6)             iterate through the properties (including event
               handlers) of the cloned widget and set them to
               those of the host widget, and make it visible
7)         if a corresponding host widget cannot be found:
8)             add this cloned widget to the parent of the
               corresponding host widget in the host GUI
               hierarchy, and make it visible
```

Using this algorithm, WADE can apply a third-party programmer's changes in the *cloned* GUI hierarchy to the GUI hierarchy of the *host* application. These changes include adding or deleting a widget, modifying the properties of a widget, or adding or modifying the event handlers of a widget.

Adding widgets is handled in the 7th and 8th statements of the algorithm. When the *Injected Add-on Manager* finds a *cloned widget* not in the *component dictionary*, it knows it is a new widget and adds it accordingly to the *host GUI hierarchy*.

Deleting widgets is implicitly handled by initially setting all *host widgets* to be invisible (1st statement in algorithm), and only making visible those found in the *cloned GUI hierarchy*. The deleted widgets therefore will remain invisible after this process, and will appear to the user as if they had been deleted from the host application. We choose to hide the widget instead of deleting it because removing a widget at runtime may be risky. As a widget may have unknown runtime dependencies, permanently removing it may cause the application to crash. Thus, we choose a safer approach to achieve a similar effect.

Property modification of a widget is also handled in a simple yet effective fashion in the 5th and 6th statements of the algorithm. The number of widgets in the host GUI hierarchy is typically not exhaustive. So, instead of expending effort to explicitly detect individual changes, we simply reset all properties of all host widgets to the properties of their corresponding cloned widgets, regardless of whether the cloned widget has been modified or not.

Event handler modifications are also implicitly handled during the property resetting process because the .NET framework treats event handlers as part of the properties of a widget. Changing and associating new program logic with *host widgets* can be effectively applied without much additional effort beyond implementing the desired functionality.

As such, we successfully integrate the GUI builder and a number of IDE features into the third-party software add-on development process.

While WADE demonstrates a promising step towards addressing the power and ease of use trade-off for runtime modifications, it is important to note that WADE is not without limitations.

Interface dynamics

WADE enables the user to perform WYSIWYG modification of the GUI hierarchy only to the initial application state. Many interfaces, however, are dynamic and rely on runtime code that may alter the interface from how it appeared at the moment it was imported into WADE (i.e., dynamic widgets). Since the content of a widget can change at runtime, content modification through the GUI editor may not be applied back to the original application. Changes to the application may also conflict with the modifications implemented in WADE, possibly leading to unstable modifications that may not behave as expected.

However, certain interface dynamics can still be addressed using the WADE approach. For example, if dynamic widgets are initialized only once upon program invocation, it may still be possible to apply modifications using a monitoring program that knows when to take action after initialization.

Custom widgets

Another limitation of WADE is that the current implementation provides limited support for modification of custom widgets. Custom widgets often have derived custom properties and behaviors that are not recognizable by the GUI builder; they therefore cannot be properly displayed in the IDE.

However, not all custom widgets are unrecognizable. Custom widgets that derive from a standard, known widget will be treated as the base widget. The GUI editor can thus handle the inherited properties, but will be ignorant of any derivative behavior and properties.

Overall, developers are advised to first get familiar with the application behavior to clearly identify customization and runtime interface dynamics before using WADE to perform runtime modifications.

Security implications

Overloading at runtime can cause problems if the replacement method violates any of the assumptions in the original application's design. It is thus recommended to practice careful and defensive programming to avoid breaking the original application logic [4].

However, as compared to toolkit modification approaches, WADE diminishes the risk of breaking the host application. In existing approaches, all modifications involve writing arbitrary code. With WADE's property editors and templates, the surface footprint of this code is diminished, and supported modifications can use known clean implementations. Writing additional code will remain risky as in Scotty and other toolkit approaches, but certain common modifications are now much safer.

USER STUDY

In order to assess the usefulness of WADE, we performed a user study. In terms of purpose and capabilities, WADE is most similar to Scotty [4]. Other alternatives, while having their own advantages, are less comparable to WADE in terms of the functionality provided or applicability. For example, surface-based approaches such as Façade and Prefab lack the ability penetrate underneath the surface; SubArtic requires use of the SubArtic toolkit to begin with.

While we expected WADE to significantly simplify add-on development as compared to Scotty, a primary objective of the user-study was to quantify the speed-up obtained with WADE over Scotty while modifying third-party software, and identify those WADE characteristics responsible for the speed-up. To this end, we performed a controlled experiment to assess and compare the strengths and limitations of the Scotty and WADE approaches.

Participants

Eight participants (7 males, 1 female) ranging from 21 to 32 years old (μ = 25.5, σ = 3.34) participated in this study. All participants were experienced computer users and programmers.

Apparatus

The experiment was conducted using a DELL Optiplex 990 Desktop computer running on the Windows XP operating system, with 4 GB RAM and Intel Core i7-2600-3.40 GHz CPU. A Dell E2211H monitor, a USB optical mouse and a standard keyboard were used as the input/output devices. The Paint.NET interface to be modified is implemented in C# using Microsoft Visual Studio.

As Scotty was developed for the Cocoa framework in Mac OS whereas WADE runs on the .NET framework in Windows, we created a Scotty-like development environment (Scotty simulator) to support user tasks on Windows 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.

ManagedSpy: a Microsoft utility program that allows developers to spy on an application's GUI at runtime. Figure 5 presents a screen-shot of the program which allows a user to discover the names, types, and properties of the host application's GUI components at runtime. The ManagedSpy serves a similar functionality to the hierarchy browser, widget picker, and object observer tools offered in the Scotty environment. For WADE, we provided the add-on manager and the WADE IDE based on SharpDevelop 4.2 with GUI builder as previously described.

Task and Training

Before the actual experiment, each participant was given a tutorial demonstration and three practice tasks similar to the experimental tasks to familiarize him/herself with the use of the Scotty simulator and WADE. For each approach, we provided a manual with the information necessary for the users to complete the tasks.

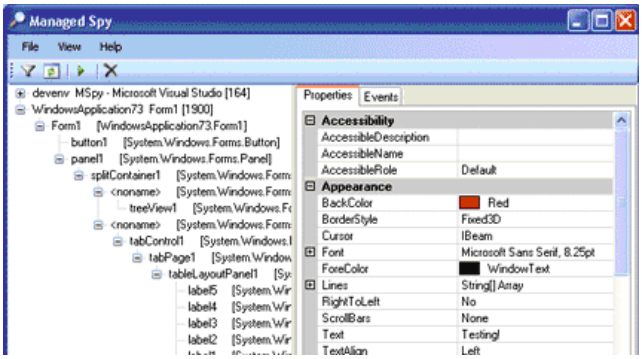

Figure 5. Screenshot of the ManagedSpy tool.

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) coding snippets to hide items, (iv) coding snippets to add 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) re-inject this DLL back to the host program.

Note that the instructions we provided made code-based modifications (as with the Scotty simulator) much easier, because in real world scenarios, the methodology for achieving GUI modifications is not obvious and must be figured out in a trial and error fashion. However, to facilitate participants' completion of the tasks, we provided all the requisite information in the user manual.

The tasks to be completed using (a) our Scotty simulator and (b) WADE in the experiment are described below:

- **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 button called "Undo all" on the icon bar (as in Figure 1). Once the "Undo all" button is clicked, it would undo all user modifications for a particular session.

Experimental Design

We used a within-participants design in which all participants were asked to perform all tasks using both approaches. Participants were randomly assigned to two groups of four participants each. Half of the participants performed the two tasks with the Scotty simulator first, followed by WADE, while the other half performed the two tasks in the reverse order. Each participant performed the entire experiment in one sitting lasting 1-2 hours, with optional breaks between tasks.

In summary, the design was as follows (excluding practice tasks): 8 subjects × 2 coding approaches (Scotty-simulator vs. WADE) × 2 tasks (GUI reconfiguration, add-on development) = 32 tasks in total. Comparative factors were time spent on the tasks, whether or not the task was successful, and participants' subjective preferences in their post-experiment questionnaire.

Results

The user-study results confirmed that software modification is much easier with WADE than with Scotty.

Accuracy: Seven participants finished all tasks, while one participant only finished the first task using both approaches. Therefore, from the task completion point of view, there was no difference between the two approaches.

However, there was a difference in the number of attempts it took for participants to finish each task. An *attempt* denotes each instance a participant believed the task was complete, and tried to execute the modifications he/she had made. Errors in program execution, therefore, resulted in multiple attempts. On average, participants required 1.13 attempts to complete a task using WADE, and 1.75 attempts with the Scotty-like approach. A paired *t*-test comparison between the two approaches revealed that this difference is marginally significant (t_7 = 4.07, p = .083). This result suggests that users are likely to commit fewer mistakes during interface modification using WADE than Scotty.

Time to task completion: We then conducted a 2x2 repeated measures ANOVA on the task-completion times with the approach type (WADE/Scotty) and task type (reconfiguration/add-on integration) as the relevant factors. Figure 6 presents the results. As expected, we found a significant main effect of the approach used ($F_{1,7}$ = 31.41, p<0.01) on the task-completion time, which implies that on average, users completed the two tasks significantly (about 2.4 times) faster using WADE (264.4 s) than with the Scotty-simulator (639 s).

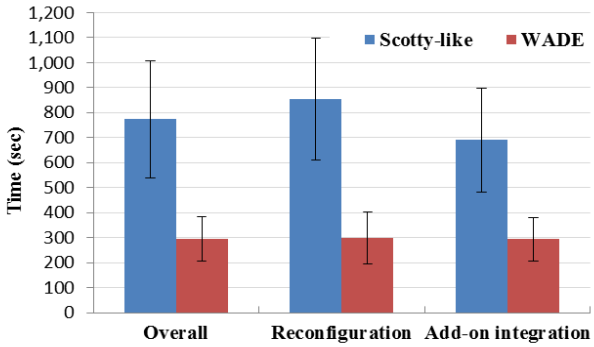

Figure 6: Comparison of task completion times for WADE and Scotty.

Qualitative comparison: After the experiment, participants were asked to rate various aspects of the two approaches on a 5-point Likert scale. In all, they answered four questions concerning *usefulness* (how useful was the software modification tool?), *user productivity* (how much did this tool improve your productivity?), *learnability* (how easy was it to learn the steps involved in this approach?) and *overall satisfaction*. 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*.

Discussion

Factors contributing to WADE's performance advantage

Results of the user study clearly demonstrate the advantages of using WADE's integrated approach for reconfiguration and add-on development tasks. The performance gain with WADE arose due to a number of factors as enumerated below.

1) The WYSIWIG GUI editor allows participants to more directly interact with and manipulate widgets and their properties. This consequently saves time and effort required to look up the GUI widget hierarchy for appropriate names and properties before applying any changes, as indicated by our participants: "*WADE enables direct manipulation which is easy, faster and intuitive. (P1, P5)".*

2) Fewer task completion attempts using WADE can be attributed to the fact that direct interface manipulation essentially involves *recognition* of widgets and their properties, while coding relies on *sense making* and *recall.* It is easier to make mistakes using the pure coding approach, as indicated by the higher average number of attempts mentioned earlier.

3) Although coding is necessary to add/modify GUI functionality, the WADE IDE provides scaffolding in the form of event handler templates to aid the development

process. "*The event handler template makes coding easier*" (*P1*).

4) In the Scotty-like approach, the sense-making process and coding for the add-ons are separate tasks handled using different tools and applications, causing additional overhead both cognitively, in terms of remembering and linking the information, as well as physically, in terms of operating and interacting with multiple, different tools. In WADE, the IDE provides an integrated environment for coding, which can reduce the time spent on managing and interpreting the code. As indicated by P4: "*Switching back and forth between ManagedSpy and IDE is tedious and frustrating*".

5) Finally, as all necessary instructions required for modifying UI components using the Scotty simulator, typically unavailable in the real world, were provided to users, latency involved in discovering the correct modification commands is not accounted for in this study. Therefore, one can expect WADE to enable an even larger performance gain over toolkit-based deep approaches such as Scotty in real-world scenarios.

In summary, the advantages of WADE over Scotty-like approaches are (1) Direct and easy location-cum-manipulation of target widgets due to the WYSIWYG editor; (2) Fewer chances of committing errors during interface modification as the UI modification process is simplified by the WADE IDE; (3) Scaffolding provided by WADE for incorporating add-ons, in the form of event handlers, enables easier and faster addition/modification of functionality; (4) Facilitation provided by the IDE significantly reduces switching time between different applications and tools; (5) Less search time required to find the correct statements to manipulate GUI properties.

While the user study conclusions are not surprising, as WYSIWYG GUI editing is easier than explicit code hacking, it demonstrates that an IDE greatly simplifies UI modification as compared to a Scotty-like approach even for relatively experienced programmers. All of our participants mentioned that they are less likely to use the Scotty simulator for implementing third-party add-ons. On the other hand, WADE significantly lowers the knowledge barrier for developing third-party GUI add-ons. Six out of eight participants indicated that they would use WADE to write add-ons for third-party software.

Limits of the GUI builder metaphor

While many of the modifications were easier to perform using a GUI builder, participants also found it less convenient when dealing with repetitive or looping tasks. For example, if a participant is asked to change 6 out of 7 labels to a different font type, it is easier to use a loop than manually perform the changes multiple times. The GUI metaphor delivers important benefits to learnability, memorability, and error prevention, but it can be inefficient for frequent users [8]. In such cases, a command language may be preferred as it allows simpler programming of similar and repetitive tasks, but at the cost of requiring the user to learn command names and syntax, putting more demands on the user's memory and increasing the chance of errors. Combining both approaches may mitigate this trade-off. For example, Inky [8] allows for sloppy command input and provides rich visual feedback to reduce the cost on user's memory, making it less error-prone. Sikuli [1], on the other hand, enables inclusion of visual images in the command to make it easier and more intuitive to refer to graphical elements. To some extent, WADE follows the same approach by introducing the GUI builder into Scotty's command line programming environment to improve the ease of use and robustness of the third party add-on development. However, our user study has revealed that there is room for improvement to better combine the advantages of the GUI builder and command line programming to further improve the efficiency and ease of use of third party add-on development.

EXTENSION TO OTHER FRAMEWORKS & PLATFORMS

Although WADE is currently only implemented for the *Windows Forms* framework, its approach can be generalized to most other frameworks and platforms.

In general, the WADE approach involves the following three framework-dependent steps:

1) Create an injected add-on manager that can enter the runtime process to manage add-ons, retrieve the GUI hierarchy information, and apply changes back to the host application.

2) Identify a suitable IDE that has GUI builder support and allows add-on integration.

3) Implement an add-on for the IDE that can import the GUI hierarchy from the host application, display it in the GUI builder, and compile the changes to a DLL add-on.

Choosing runtime code intervention method

The key to step 1 is runtime code observation and intervention. On Windows, we use DLL injection. (A solution for Mac OS X is described in [4].) While there are several ways to achieve DLL injection in Windows, we present two primary methods below: a) registry key-based injection and b) system hook-based injection [14].

Registry key-based injection works by adding a new DLL to a registry AppInit key. In Windows Vista and Windows 7, this feature is disabled by default, but can be achieved through code signing. Whenever a new application loads, the DLL will be loaded into the same process as well.

System hook-based injection works by using a separate background monitoring application that detects new programs and uses methods such as SetWindowsHookEx. While more cumbersome and complex, this approach injects the DLL at the deeper thread level and can be used by all versions of Windows.

Choosing which method to use depends on the frameworks used. Some (*e.g.*, Windows Forms) allow modification of the UI thread in the process level. Other frameworks (*e.g.*, QT [11]) do not allow such modifications; therefore, thread level intervention becomes necessary. Once the appropriate runtime code observation and intervention method is identified for a particular framework and platform, the remaining effort mostly concerns the work of writing the injected add-on manager for the framework.

Identifying IDEs with GUI builder and add-on support

The second step is to choose a suitable IDE that supports GUI editing for add-on development. To shorten the development time, it is recommended that an existing IDE be chosen for a particular framework to work.

As WYSIWYG GUI editing becomes more popular, it is not difficult to identify such IDEs for many of the modern frameworks. For example, in the Java platform, NetBeans and Eclipse are two such IDEs; Qt Creator [10] is an example that satisfies these requirements for the popular Qt framework; XCode is an IDE that is suitable for the Mac OS Cocoa framework. We implemented the WADE prototype for both the Visual Studio and Sharp Develop IDEs.

Developing an add-on for importing and presenting GUI

Once a suitable IDE is identified, the steps mentioned in the implementation section can be followed to create an add-on that can import and present the host GUI hierarchy in the IDE's GUI builder. The exact process of implementing add-ons may be complex and depends on the details of the particular environment. However, it is technically feasible and the approach we have proposed in the implementation section can serve as a useful guideline for the development process.

CONCLUSIONS AND FUTURE WORK

The WADE IDE is shown to be useful for realizing a variety of GUI-based modifications in existing software. The presented user study confirms that while these modifications are achievable employing alternative approaches, WADE significantly lowers the requisite knowledge and effort barriers. Future work involves extending the current implementation to other OS platforms, widening WADE support to handle custom and dynamic widgets, and potentially enabling debugging capabilities inside the WADE IDE for add-on development.

ACKNOWLEDGMENT

We thank the AC and anonymous reviewers for their constructive comments and feedback. We thank members of the NUS-HCI Lab for their support. This research is supported by National University of Singapore Academic Research Fund WBS R-252-000-414-101 and by A*STAR, Singapore, under the Human Sixth Sense Program (HSSP) grant.

REFERENCES

1. Besacier, G., and Vernier, F. Toward user interface virtualization: legacy applications and innovative interaction systems. In EICS, 157–166, 2009.
2. Dixon, M., and Fogarty, J. Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In CHI, 1525-1534, 2010.
3. Dixon, M., Leventhal, D., and Fogarty, J. Content and hierarchy in pixel-based methods for reverse engineering interface structure. In CHI, 969–978, 2011.
4. Eagan, J.R., Beaudouin-Lafon, M., and Mackay, W. E. Cracking the cocoa nut: user interface programming at runtime. In UIST, 225–234, 2011.
5. Edwards, W. K., Hudson, S. E., Marinacci, J., Rodenstein, R., Rodriguez, T., and Smith, I. Systematic output modification in a 2d user interface toolkit. In UIST, 151–158, 1997.
6. Grigoreanu, V., Burnett, M., Wiedenbeck, S., Cao, J., Rector, K., and Kwan, I. End-user debugging strategies: A sensemaking perspective. In TOCHI, 19(1):5:1–5:28, May 2012.
7. Mackay, W.E. Triggers and barriers to customizing software. In CHI, 153–160, 1991.
8. Miller, R.C., Chou, V.H., Bernstein, M., Little, G., Kleek. M.V., Karger, D., and schraefel. M. 2008. Inky: a sloppy command line for the web with rich visual feedback. In UIST (2008), 131-140.
9. Pirolli, P. and Card, S. The sense-making process and leverage points for analyst technology as identified through cognitive task analysis. In ICIA, 2005.
10. Qt Creator. http://gitorious.org/qt-creator/qt-creator
11. Qt framework. http://qt-project.org/
12. Robinson, M. Design for unanticipated use. In ECSCW, 187–202, 1993.
13. Shewmaker, J. Analyzing DLL Injection, http://www.bluenotch.com/files/Shewmaker-DLL-Injection.pdf.
14. Stuerzlinger, W., Chapuis, O., Phillips, D., and Roussel, N. User Interface Facades: Towards Fully Adaptable User Interfaces. In UIST, 309–318, 2006.
15. Tan, D.S., Meyers, B., and Czerwinski, M. Wincuts: manipulating arbitrary window regions for more effective use of screen space. In CHI extended abstracts, 1525-1528, 2004.
16. WineHQ. http://www.winehq.org/.
17. Yeh, T., Chang, T., and Miller, R. Sikuli: using gui screenshots for search and automation. In UIST (2003), 183-192, 20