Retrofitting Collaboration into UIs with Aspects

Li-Te Cheng, Steven L. Rohall, John Patterson, Steven Ross, Susanne Hupfer

IBM Research
Collaborative User Experience Group
Cambridge, Massachusetts

{li-te_cheng,steven_rohall,john_patterson,steven_ross,susanne_hupfer}@us.ibm.com

ABSTRACT

Mission critical applications and legacy systems may be difficult to revise and rebuild, and yet it is sometimes desirable to retrofit their user interfaces with new collaborative features without modifying and recompiling the original code. We describe the use of Aspect-Oriented Programming as a lightweight technique to accomplish this, present an example of incorporating presence awareness deeply into an application's user interface, and discuss the implications of this technique for developing CSCW software.

Categories and Subject Descriptors

D.2.13 [Software Engineering]: Reusable Software – reuse models; H.5.3 [Information Interfaces and Presentation]: Group and Organization Interfaces – Computer-supported cooperative work, collaborative computing

General Terms

Design, Human Factors, Languages, Theory.

Keywords

CSCW, groupware, aspect-oriented programming, application retrofitting, user interface components.

1. INTRODUCTION

It is often desirable to introduce collaborative features, such as instant-messaging and email, into the user interface of software (e.g. "contextual collaboration" is an approach that embeds new collaborative capabilities into familiar non-collaborative applications [6]), but some applications are not amenable to revision or reconstruction. Retrofitting collaborative features into legacy systems, in-house/custom-built software, and mission-critical applications using conventional approaches may be too expensive, time-consuming, and risky to be worthwhile.

Ideally, the retrofitting process should have as little impact on the application as possible, yet it must incorporate the desired set of collaborative features. In this paper we list the options for retrofitting, and focus on one promising strategy that embodies the best qualities of these options: Aspect-Oriented Programming [7]. We summarize the minimal set of concepts from Aspect-Oriented Programming needed for retrofitting, and present a working example of an application retrofitted with this technique to enable presence awareness, and discuss the implications of this technique for developing CSCW software applications.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CSCW'04, November 6-10, 2004, Chicago, Illinois, USA.
Copyright 2004 ACM 1-58113-810-5/04/0011...$5.00.

2. LEVELS OF RETROFITTING

Retrofitting can occur at three levels – the application level, the programming environment level, and the operating system level. Each level has its own set of options, strengths, weaknesses, and CSCW-related examples for the application developer to consider.

Retrofitting at the application level enables the developer to leverage any extensibility offered by the application's architecture. The chief benefit is that any collaborative features that are introduced will exist gracefully within the application. Ideally, the framework for extension would focus on the application-specific issues and insulate the developer from peripheral and low-level details of the operating environment around the application.

Examples include using application programming interfaces intended for third-parties to hook in new components (e.g. Churchill et al. use Microsoft ActiveX application interfaces to anchor chats inside Word [3]), or creating a proxy service to intercept and change the standardized protocols for communication and presentation supported by the application (e.g. SmartPrinter used a proxy leveraging the printing protocol used by all applications in their workplace to insert awareness information on printouts [5]). However, the original architects of the applications cannot be expected to foresee every future contingency, and the available application programming interfaces and standard protocols may be limited or nonexistent.

Retrofitting can also be considered at the programming environment level: one may be able to exploit the runtime characteristics of the environment used to create the application – particularly the dynamic capabilities of the language for the component responsible for the user interface. Some programming language environments are flexible, and offer options for programs to modify themselves at runtime and dynamically load new modules, without requiring recompilation. The main benefit here is the potential to significantly customize the behavior of the application beyond the original design of the application.

Other environments offer some flexibility in manipulating the language's runtime libraries for user interfaces and event handling, without rebuilding the entire application. For example, through a custom class loader, Flexible JAMM does runtime replacement of Java's single-user interface components with collaborative equivalents [2].

A problem with this approach is that not all programming environments have the needed flexibility. The application being retrofitted may be coded in a restrictive environment, and have requirements for strict control over runtime configuration that may deny modification access to certain runtime libraries. Also, if well-defined APIs are not available, it may be difficult to customize or introduce new behaviors into the application. For example, while it might be easy to replace the default label widget with a new one by replacing the widget library at runtime, specifying that only one particular label use the customized label might not be possible with this technique.

The final level to consider for retrofitting involves diving into the operating system to trap event calls, capture pixels on the screen, and hook into the boundary between the application and the operating system's services. A significant advantage of this option is the ability to treat the application like a "black box". This is especially useful for old applications whose documentation and source code may be long lost. Another consequence of this "application independence" is that the techniques used to retrofit one application may work for another.

Many typical application-sharing systems take this approach (e.g. VNC [11]), enabling them to share entire applications on the desktop. Li and Li, in their survey and in their own system, discuss how to extend this approach to share only selective pieces of state [10]. There are a number of drawbacks to the operating system level approach. While the application becomes a "black box," the developer must now focus on the intricacies of the operating system. The deep semantics and data structures of the application are also obscured; only events and visible elements of the user interface are discernable at the operating system level. Moreover, there may be interference from events and side-effects from other applications and services running in the operating system. Thus, intelligent analysis of these discernable events may be required for seemingly simple application operations.

Each of these levels highlights a diverse array of examples and suggests desirable characteristics to help retrofitting. The application level spotlights access to the deep semantics of the application through clearly defined programming interfaces. The programming level points out the flexibility afforded by modifying runtime configurations. The operating system level showcases the richness of trapping events.

3. ASPECT-ORIENTED PROGRAMMING

Aspect-Oriented Programming, or AOP for short, is an approach that draws upon the three desirable qualities for retrofitting as discussed in the previous section. We first provide a brief introduction to AOP, and elaborate its specific features that benefit retrofitting.

"Aspects" are special objects that define rules for actions occurring before, after, and within code. While Object-Oriented Programming is a methodology for software modularization, where specific pieces of application functionality are separated into objects, AOP extends this separation further, by effectively modularizing calls within objects that are being repeated across disparate objects into aspects.

A major benefit of this approach is a separation of secondary, supporting functionality (now expressed as aspect objects) from the core objects of the application. The core code becomes simply focused on the core requirements. The rules in the aspects automatically apply the secondary functionality at runtime. See Kiczales et al [7] and Laddad [9] for more detailed explanations of AOP and examples.

In the case of retrofitting, the objects in the application being retrofitted represent core functionality, and the collaborative features being introduced would be represented by one or more aspects. These "collaborative" aspects contain rules indicating where to retrofit their capabilities into the application.

AOP adds new language-agnostic concepts and has been implemented in many languages. Three concepts are relevant for retrofitting: defining and instancing aspects, specifying rules, and integrating aspects with existing code.

Aspects are declared similarly to how classes are declared in the host programming language. For the most part, they are no different from any other object in an object-oriented application, and have attributes, methods, inheritance, etc. The main difference is the incorporation of rules, and how they are instanced. Aspects are not instantiated programmatically – they only appear when their rules are triggered at runtime.

The rules that tie an aspect to other objects in an application are defined by conditions, termed *pointcuts*, and actions, termed *advice*. From a retrofitting perspective, creating rules in aspects is akin to monitoring for desired patterns of events from the targeted application. *Pointcuts* actually refer to points or regions of program execution, which can be expressed as a variety of object operations, including private or public method calls, object instantiations, attribute assignments, scoping conditions, and program flows. *Pointcuts* can even refer to private calls and attributes, thus exposing the internal programming interfaces of the application. Thus, we can leverage inner application semantics in addition to events passed between the application and the runtime environment. *Pointcuts* can also declare context to capture data from the associated pieces of program execution, such as parameters passed into methods and the calling object.

Advice are associated with *pointcuts*, bring in context around *pointcuts*, and specify when to apply actions when *pointcuts* are encountered. An *advice* is where collaborative features get established and invoked in the application.

Finally, there are two ways to introduce aspects into an application. The first approach is to use an aspect compiler that compiles the aspect and generates hidden intermediate objects that express the aspect in the original language of the application. The intermediate objects use reflection and event hooking to ensure that *pointcuts* are established with appropriate *advice* into the application, without recompiling the original code. The end result is a self-sufficient application whose code appears to be in the original language of the application. The second, known as "runtime weaving," is to use a special runtime that dynamically incorporates the aspects with the targeted application during execution. These two options are examples of the flexible runtime configuration characteristic for retrofitting – depending on the application requirements; one approach may be more suitable than the other.

4. ADDRESS BOOK + BUDDY LIST

We now illustrate an aspect-based technique of retrofitting collaborative features into an application.

We took a basic address program that was an example Java application, which we did not write, using the SWT widget library [4]. The program is a single-user application that lets the user enter contact information, save and load all contact data, and conduct searches. A screenshot is seen at the top of Figure 1.

The address book can benefit from awareness information provided by an instant messaging service. Our final result can be seen on the bottom of Figure 1. We have the same application, but now names are decorated with icons denoting online status such as online, away, and do not disturb. Tooltips over the names reveal a detailed status message. We did not have to recompile the original application, and only added one aspect written in about one hundred lines of code that interacted with the IBM Lotus Sametime instant messaging toolkit [8].

Our strategy to accomplish the retrofit was threefold. First: understand the application from its runtime behavior and its codebase, looking for useful internal application programming interfaces. Second: identify the *pointcut* and define *advice* where we can initialize the new collaborative feature upon application startup. Third: identify the *pointcut* and define *advice* where we can establish a foothold into the user interface and add to it.

From understanding the operation of the address book application, we learned that the address book is represented by an AddressBook class, which includes an open() method that is called when the application is starting up, and returns an SWT Shell object (the widget for the entire application window). Also, the address book uses a Table widget consisting of TableItem widgets for each row. Each TableItem contains the fields for one contact, which are set up using a setText() method. The key field is the email address which we can use as an identifier to get online status information from the instant messaging service. There are also unused methods to set icons and tooltips in the table.

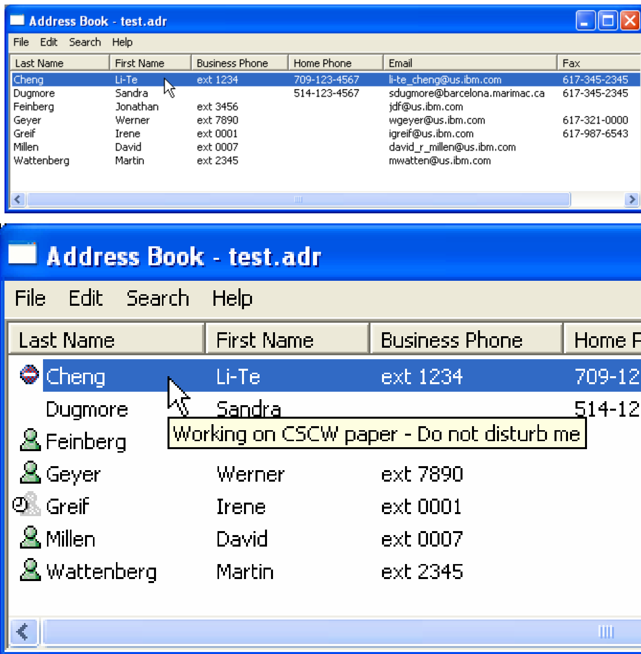

Figure 1 –Original address book application (top) and the same application retrofitted with IM presence icons and tooltips (buttom)

We then used AspectJ [1], which provides extensions to Java for AOP, to define an aspect representing instant messaging awareness information associated with a row entry in the table of the address book. Figure 2 shows the aspect we created, with the internals written in pseudo-code for brevity. Sections A and C in Figure 2 define the *pointcuts* of interest. Sections B and D define the *advice* corresponding to the *pointcuts* in sections A and C respectively.

Section A defines a *pointcut* on any call to the open() method of the AddressBook class. This captures the moment when the application is starting up. This is an important moment to allow us to set things up related to our new collaborative feature.

Section B defines the *advice* using section A's *pointcut*. The "after" keyword specifies that the *advice*'s actions will execute after the *pointcut* is completed (i.e. after the open() method returns with something). The "returning(Shell shell)" piece allows the *advice* to capture the return value from the call to the open() method. The returned shell widget gives a parent widget in which to pop up a login dialog when the *advice* is triggered. After obtaining the login information from the dialog, we log into the instant messaging service, and set up a listener for status changes. Upon status changes, the listener updates the icon in the appropriate row of the table using a hashtable that maps email addresses to rows.

Section C defines a *pointcut* on any call to the setText() method of the TableItem widget. This specifies the moment when a row with contact information is being created or changed. This is an important moment to set up an awareness icon in the table, and establish tooltip information. The "target(item)" piece of

```
public aspect LiveName
{
  after() returning(Shell shell):                          (A)
    call(* AddressBook.open(..))
  {                                                        (B)
        // Display login dialog in shell
        // Login to IM System
        // Add listener for IM status changes, update icons
  }

  after(TableItem item) :
      (target(item) &&                                     (C)
       call(* TableItem.setText(..)))
  {
        // Get email from table item
        // Get current IM status using email               (D)
        // Get icon based on IM status
        // Add listener for mouse hover, show IM status text
        // Add icon to table item
  }
}
```

Figure 2 – Pseudo-code showing the aspect responsible for retrofitting the Address Book application

the *pointcut* captures the actual TableItem widget instance calling the setText() method and associates it with the “item” parameter.

Section D defines the *advice* using Section C’s *pointcut*. Again, the “after” keyword specifies the actions for this *advice* that are invoked after the setText() method is completed. The “item” parameter from the *pointcut* is passed through to the *advice*. The code in the *advice* extracts the email field from “item”, which is then used to query the instant messaging system for status information. The status information is then mapped to appropriate tooltip and icon information to display in the table (“item” allows us to access the appropriate methods). The hashtable used in section B is updated with a mapping between the email address and the table row.

We then compiled our new aspect, linking in the instant messaging library and the binary for the address book application. The final result was an application that operates largely the same as before, but with a new feature. We did not need to change or rebuild the original application code.

This implementation can be improved, e.g. we could specify much more in our aspect. For example, we could extend the existing context menu with an option to start a chat conversation from a name. Structurally, instead of concentrating all of our new functionality in the aspect code, we could also define regular non-aspect classes encapsulating the awareness functionality, and reduce our *advice* actions to calls to these classes. This way, the aspect is focused on bridging the address book application’s objects and the objects associated with the new features.

5. CONCLUSIONS

In this paper, we used AOP to retrofit applications with collaborative features, and presented an example where we did not have to recompile the original application code.

Unlike other approaches operating at the programming language level of retrofitting [2], we operated within the confines of the programming interfaces of the original application, reusing their existing user interface widgets instead of replacing them entirely. Also, using AOP, we were able to write code at the application level and introduce changes in carefully-selected sections of the application. Like retrofitting at the operating system level, we can monitor the application’s event flow through *pointcut* definitions, but we can use them to take advantage of inner semantic context such as return types, calling objects and passed parameters, and invoke *advice* that leverage the internal programming interfaces.

Thus, retrofitting with aspects combines desirable qualities found at the application, programming, and operating system levels. The new code feels native to the application, leveraging existing resources and programming interfaces. However, this technique’s effectiveness hinges upon the first step of our strategy: understanding the application. Without *a priori* knowledge about the application’s workings, it would be difficult to express the *pointcuts* to inject new behavior at the appropriate moments of the application’s operation. Available source code, standards, APIs, decompilers, and tracers are very useful means of understanding the application. But some applications may be truly opaque, or too large and complicated to analyze. This is a problem already faced by retrofitting at the operating system level, and there is an opportunity for automated and intelligent tools to help [10].

We only retrofitted at the user interface layer of the application, but aspects can benefit other layers as well. One strategy is to use aspects as retrofitting “bridges,” with minimal code that binds the core application and other frameworks and toolkits specialized for collaboration infrastructure as well as user interfaces.

Finally, there are other types of collaborative features to explore besides the example we described here. Enabling synchronous sharing of applications, adding networked peer-to-peer file sharing, introducing roles, policies, and access control, and suppressing, altering, and rerouting the flow of user interface events are some collaborative features that may be amenable to aspect-oriented retrofitting.

6. REFERENCES

[1] AspectJ. http://www.eclipse.org/aspectj.

[2] Begole, J., Rosson, M., Shaffer, C. Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems. *ACM Trans. on Computer-Human Interaction*, 6, 2 (June 1999), 95-132.

[3] Churchill, E., et. al. Anchored Conversations: Chatting in the Context of a Document. In *Proc. of the SIGCHI Conf. on Human Factors in Computing Systems (CHI 00)* (The Hague, Netherlands, April 1-6). ACM Press, New York, NY, 2000, 454-461.

[4] Eclipse.org. http://fullmoon.torolab.ibm.com/downloads/drops/R-2.1.2-200311030802/

[5] Grasso, A. and Meunier, J-L. Who Can Claim Complete Abstinence from Peeking at Print Jobs? In *Proceedings of the 2002 ACM Conf. on Computer Supported Cooperative Work (CSCW 02)* (New Orleans, USA, Nov. 16-20). ACM Press, New York, NY, 2002, 296-305.

[6] Hupfer, S, Cheng, L., Ross, S., Patterson, J. Introducing Contextual Collaboration into an Application Development Environment. In *Proceedings of the 2004 ACM Conf. on Computer Supported Cooperative Work (CSCW 04)* (Chicago, USA, Nov. 6-10). ACM Press, New York, NY, 2004.

[7] Kiczales, G., et. al. Aspect-Oriented Programming. In *Proc. of European Conf. on Object-Oriented Programming (ECOOP 97)* (Jyväskylä, Finnland, June 1997). Springer-Verlag, Germany, 1997, 220-242.

[8] IBM, Sametime Java Toookit. http://www-136.ibm.com/developerworks/lotus/products/instantmessaging

[9] Laddad, R. *AspectJ in Action*. Manning, Greenwich, CT, 2003.

[10] Li, D. and Li, R. Transparent Sharing and Interoperation of Heterogeneous Single-User Applications. In *Proc. of Computer Supported Cooperative Work (CSCW 02)* (New Orleans, USA, Nov. 16-20). ACM Press, New York, NY, 2002, 246-255.

[11] RealVNC. http://www.realvnc.com