Cracking the Cocoa Nut: User Interface Programming at Runtime

James Eagan, Michel Beaudouin-Lafon, Wendy E. Mackay

Cracking the Cocoa Nut: User Interface Programming at Runtime

James R. Eagan[1,2]
eaganj@lri.fr

Michel Beaudouin-Lafon[1,2]
mbl@lri.fr

Wendy E. Mackay[2,1]
mackay@lri.fr

[1]LRI (Univ. Paris-Sud & CNRS); [2]INRIA
91405 Orsay Cedex, France

ABSTRACT
This article introduces *runtime toolkit overloading*, a novel approach to help third-party developers modify the interaction and behavior of existing software applications without access to their underlying source code. We describe the abstractions provided by this approach as well as the mechanisms for implementing them in existing environments. We describe Scotty, a prototype implementation for Mac OS X Cocoa that enables developers to modify existing applications at runtime, and we demonstrate a collection of interaction and functional transformations on existing off-the-shelf applications. We show how Scotty helps a developer make sense of unfamiliar software, even without access to its source code. We further discuss what features of future environments would facilitate this kind of runtime software development.

Keywords: user interfaces, meta-toolkits, runtime software development, runtime toolkit overloading

ACM Classification: H.5.m Information interfaces and presentation (HCI): Misc.

General terms: Human Factors

INTRODUCTION

Software designers create software with a particular model in mind of how it will be used. Even in the most thoughtfully designed systems, however, these assumptions may break in the face of users' own constraints and demands. Assumptions made by a designer, based on a deep understanding of hardware, software, and human constraints, may change. As these constraints evolve, so too must the software, often in ways not easily reconcilable with its original design.

Even when these constraints do not evolve, users frequently push their software in ways not anticipated by its designers [20]. A user's interaction is strongly situated in a particular context, with particular needs that may arise and disappear spontaneously and in unforeseeable ways [22], even with the best design practices. As such, there is a gap between the designers' conception of how software will be used and its actual use. These gulfs are likely to continue to grow as software is used in an increasingly rich, ubiquitous, and distributed environment.

Different approaches have been taken to bridge these gulfs by making software more flexible, including giving users more control over their software's configuration [16], runtime behaviors [18, 8] and interfaces [21, 9], or by giving programmers more control over their language [15]. These approaches all build on the notion that, although the designer may have a good understanding of how to create software that supports the most common use cases, users will push their software in unintended ways.

Our goal is to bridge the gap between the designed use and actual needs of software by enabling runtime software development. Runtime software developers are third-party developers who can modify the behavior and/or interface of software that nearly satisfies the user's needs. Runtime development is concerned with taking the existing behavior of an application and changing it in some way, without requiring access to the application source code.

In this paper, we show how to facilitate this kind of runtime software development in existing applications, we show how it can facilitate new interactions, and we argue for changes to better support it. In particular:

- We present *runtime toolkit overloading*, a novel approach that provides high-level abstractions to make the redefinition of program behaviors a first-class operation, without access to the the application's source code;
- We describe how to implement this approach using a set of deep hooks into existing toolkits and runtime systems, taking advantage of modern programming languages;
- We present *Scotty*, an implementation of runtime toolkit overloading in the Mac OS X Cocoa environment, and illustrate its use to modify a set of off-the-shelf applications;
- We provide recommendations to improve support for runtime development in future user interface toolkits.

RELATED WORK

Some applications are designed with modification as an explicit design concern and thus offer some kind of plugin,

UIST'11, October 16-19, 2011, Santa Barbara, CA, USA.

scripting, or extension mechanism. Most applications, however, are not designed for such extensibility. For these kinds of applications, other methods are necessary. In this section, we examine different approaches to black-box modification of programs (Figure 1). These approaches are black-box in the sense that they do not assume that the runtime developer has any access to the program source code or to its internal structures.

Scripting, Plugins, and Extensions

Many applications support some degree of extensibility by users through scripting interfaces and/or plugin and extension mechanisms. Almost all Mac OS applications, for example, offer a scripting interface (through AppleScript) that gives control over standard functionality such as launching, quitting, and moving windows [3]. Many web browsers, including Safari, Firefox, and Chrome, offer plugin and extension interfaces, allowing one to write modifications to the way the browser loads and displays web pages. Scripting and plugin interfaces can provide varying degrees of control, depending on what internal structures of the application they expose and what, if any, sandboxing they perform to insulate the plugin code from the rest of the application. For example, the Firefox browser is open enough to enable the Chickenfoot [7] plugin to provide a domain-specific scripting language with deep access to web pages. Similarly, Google's Chrome Frame[1] uses Internet Explorer's plugin system to replace the rendering engine with that of the Chrome browser.

These interfaces can potentially offer a high degree of power, but only if the application developer has included explicit support for them. Moreover, for complex changes to the program behavior, it is typically necessary to create *ad hoc* modifications on a per-application basis. Thus, they may be of limited utility for defining new behaviors for all applications. In order to support runtime software development, we want a general mechanism to write plugins for any application, regardless of whether it was designed with such support in mind.

Changing Program Behavior

When the application does not provide explicit support for runtime modifications, other approaches are necessary. The most basic of these approaches is to operate on the surface representation of a window. The surface representation is the part of the application directly exposed to the user, such as the collection of pixels on the screen, but it could also include auditory, haptic, or other modalities.

Surface Representations A typical example of this approach is VNC [19], which teleports the pixels of an entire desktop from one machine to another, allowing the user to remotely use the entire desktop. While such remote-use capabilities have long been embedded in the design of some windowing systems (*e. g.* X Windows), this approach grafts runtime capabilities into the desktop environment, allowing arbitrary applications to be used remotely. WinCuts [23] uses a similar approach to allow the user to create proxy windows based on just a relevant region of a window. Both of these approaches provide a user with the means to change the display of existing applications at runtime. They rely on the fact that the modification environment does not need to know about the deeper semantics of the application. It suffices to simply transport or transform the display and let the user make sense of it. Because no semantic information is available, however, the transformations are therefore limited.

[1] `www.google.com/chromeframe`, 20 April 2011.

	WinCuts	Façade	Prefab	SubArctic	Scotty
Access to Surface Rep	√	√	√	√	√
Can change surface rep.		√	√	√	√
Access to widgets		√	√	√	√
Can change widgets		(via I/O redirection)	(via I/O redirection)	√	√
Access to program model				(only parts that touch toolkit)	√
Can change program model				(only parts that touch toolkit)	√
Works with multiple platforms	√		√		
Modifications are typically "safe/ robust"	√	√	√		

Figure 1: Summary of the capabilities and characteristics of different approaches to runtime modification.

Augmented Surface Representations It is possible, though, to augment the surface representation with supplemental semantic information. Façade [21], for example, integrates with the X Windows system to gather information about the widget hierarchy. With this additional information, Façade is able to operate at a finer granularity than WinCuts and can even change the behavior of individual widgets.

Prefab [9, 10] takes a different approach to augmenting its understanding of the surface representation of widgets: it uses a vision-based system to recognize widgets using their visual appearance and to create visual stand-ins for them. This has the advantage of being able to operate in a multiplatform environment, simultaneously operating on, *e. g.*, native Windows widgets, Mac OS and X Windows widgets via a VNC connection, and Flash widgets embedded in a web page. Nonetheless, it can only handle widgets it has been trained to recognize.

All of these approaches face the same limitation: Even with supplemental information about, *e. g.*, the widget hierarchy, widgets are still opaque objects. The system may know, for example, that a collection of pixels on the screen corresponds to a pull-down menu with n items, but it cannot know that widget's relation to other program objects beyond what can be inferred from its surface representation.

Toolkit Integration Integrating with the underlying toolkit can provide deeper access to interactive software. Because this approach works at the toolkit level, it has access to program objects that interact with the toolkit interfaces. Modifying the toolkit can thus allow all applications built using that toolkit to use those changes. However, those changes will

only apply to applications that were built using that toolkit. Thus, changes made to Java's Swing will not apply to Mac OS or web applications and *vice versa*.

SubArctic [11] uses this approach to add output modification hooks to Java's AWT. It uses a combination of subclassing and wrapping to make the graphics capabilities of SubArctic applications extensible. With these hooks, developers can write new modules that transform the way objects are drawn. The design of the toolkit allows these new modules to be added without explicit support by the application developer.

JAMM [5] uses a clever approach that alters Java's serialization mechanism to swap out certain classes in the serialization stream with collaboration-aware subclasses. This approach makes it possible to completely or partially replace the behavior of existing classes as long as they support serialization and have not previously been serialized. It does not, however, support the replacement of non-serializable classes or the dynamic modification of classes after serialization, since the remote end would have already received a previous version of the object's class name.

In a different area, Chromium [14] replaces the OpenGL backend of an application to automatically distribute rendering to a cluster of graphics cards and displays without recompilation. Mercator [12] uses a similar approach to infer semantic information about user interfaces in order to create auditory interfaces for blind users. The first part of this approach, like Façade, involves monitoring the communications between the application and the window server. The second part involves modifying the low-level toolkit libraries so that they communicate additional accessibility-related semantic information. While this does not require modifying the source code, it does require relinking the application.

If a toolkit is available as a shared library, however, it may be possible to avoid relinking by creating a binary-compatible stand-in for that library. This approach has been used by the WINE project[2] to create a reverse-engineered implementation of the Windows APIs for Linux. Besacier and Vernier [6] have also combined this technique with the wrapping technique used by SubArctic to create dynamic library stand-ins that partially override the behavior of a toolkit, such as replacing check boxes with crossing-enabled [1] variants.

The World Wide Web Web-based applications are built upon a combination of the HTML document object model (DOM), Javascript, and the HTTP client-server protocol. The open nature of this model lends itself to runtime modifications of web pages, in particular through browser extensions such as Greasemonkey.[3] However, any computation performed on the server remains opaque. Additionally, many programmers have taken to obfuscating their Javascript code, either to prevent human comprehension or to support code compression.

RUNTIME TOOLKIT OVERLOADING

In order to create a general solution for runtime software development, we need the power of an unrestricted plugin architecture, the generality of toolkit integration, and the ability of surface representation approaches to operate without access to a program's source code. To accomplish this, we propose a technique called *runtime toolkit overloading* that relies on dynamically loading code into an existing application. Since we do not want to require a runtime developer to reverse engineer the internal structure of every program, we provide a set of high-level abstractions that runtime developers can use to create a rich variety of modifications to an application's existing behavior. Together with the existing underlying toolkit, these abstractions form an extensible toolkit that allows a runtime developer to build on the interface and functionality already present in the application.

Hook	Description	Cocoa Method Overridden
windowCreated	Gives access to window after it has been created but *before* it appears on screen.	NSWindow -init..:
windowDidAppear	Called *after* window appears on screen.	(built-in)
windowIsUpdating	Gives access to previous window contents, just *before* redraw.	NSWindow -flushWindow
windowDidUpdate	Called *during* the rendering loop, just *after* actual drawing occurs.	NSWindow -flushWindow
windowWillHide	Gives access to window just *before* it is removed from the screen.	NSWindow -orderOut:
windowDidHide	Called with the ID of a window after it has been removed from the screen.	(built-in)

Figure 2: Window hooks available in our Scotty demonstrator. Widget hooks are similar.

Runtime toolkit overloading uses a model in which developers create *plugins* that integrate into existing applications, modifying their interface and/or behavior in arbitrary ways. These plugins use the abstractions described below to modify program behavior without having to inspect the underlying program implementation unless necessary. They are different from the plugin interfaces in many applications in that they have full, unrestricted access to the application's program space. We now describe the six abstractions that we have found useful to support runtime development.

Window and Widget Hooks In order to transform how windows and widgets appear on the screen, we need to be able to intercept and modify windows *before* they appear on the screen, such as by adding or removing widgets, changing their layout, or changing attributes such as having a title bar or being minimizable. We have defined a set of window hooks (Figure 2) that go beyond standard notifications to tap into the window creation and drawing systems. Unlike SubArctic [11], which uses subclasses to wrap the native drawing system, our approach incorporates its hooks directly into the native drawing system. In other words, rather than deriving an extensible toolkit, we transform a non-extensible toolkit into an extensible one.

Window notifications are a standard part of most UI toolkits that allow a programmer to react to window creation, update, and dismissal. These notifications, however, are not designed with program modification in mind and do not provide the necessary hooks. For example, window creation notifications are typically posted after a window has been exposed to the

[2] `www.winehq.org`, 20 April 2011.
[3] `www.greasespot.net`. 20 April 2011.

screen. Any changes made to that window, then, will appear only after the window has already been drawn once, creating an undesirable flashing effect. By contrast, our window hooks are also called *before* the window appears or disappears from the screen (Figure 2).

Similarly, window update notifications are often posted *after* a window has already been updated. These events thus occur after any window redrawing operations have been completed, precluding operations that need to be performed within an active drawing state such as those that alter the graphics context. Additionally, if these notifications are posted after redrawing has already been completed, any modification that depends on the previous state of the window will need to cache the previous contents of the window. With our hooks, however, a plugin can query and even alter the window graphics state just before the repaint is posted to the screen.

In the Web environment, applications do not have access to low-level drawing of elements. Instead, the content of the DOM tree, together with the style sheet, define how elements are drawn. Window update events therefore do not make sense. However standard DOM events track the changes to the DOM structure and therefore can be used to implement the widget appearance and disappearance hooks.

Event Funnels Many toolkits, including Java Swing, Mac OS Cocoa, and the DOM, use an event model in which interaction events are delivered through an application, window, and widget hierarchy to a callback appropriate to the event. In Cocoa, for example, a mouse press event on a widget is dispatched to its `-mouseDown:` method[4] and a mouse release event to its `-mouseUp:` method. We want to be able to intercept *all* events sent to a widget, window, or application, regardless of event type. In Cocoa and Swing, there is a single method to capture all events sent to an application, but not for all events sent to a particular window or widget.

An event funnel is a *metaclass* that intercepts any standard event (*e. g.* mouse, keyboard, tablet, cursor) and dispatches it to a single general-purpose event handler. This handler provides a way to selectively intercept all events sent to an object while maintaining the context of the object, eliminating the need for plugin developers to re-write each callback by hand and decreasing the likelihood that they might accidentally omit a stray event (*e. g.*, a tablet event). Furthermore, they relieve the programmer from having to replicate the complex event dispatch chain when redirecting events.

Glass Sheets Glass sheets are transparent layers drawn on top of whatever the software has already drawn. These sheets can be affixed to a given widget, a window, or the screen as a whole. When attached to a widget or window, the sheet will automatically reposition and resize itself as expected with its widget or window. Transparent overlays are a standard feature of the Java Swing toolkit but are not present in other common toolkits such as Mac OS Cocoa or the DOM.

Glass sheets allow the developer to create a drawing surface that is tied to a widget but independent of the widget's drawing methods. Thus, a developer can create, *e. g.*, an overlay above a video player where the overlay only redraws when it needs to change. (If the overlay does need to synchronize with drawing of the underlying view, it can use the aforementioned window hooks.) Additionally, a glass sheet that is attached to the screen instead of a particular window or widget can be used for displaying content that is independent of any widget or window's position on the screen.

[4]Objective-C purists who object to our use of the term "method" should read "message handler" in its place.

```
1 @scotty.replaceMethod(NSWindow.flushWindow)
2 def scottyFlushWindow(self):
3     try:
4         # ... trigger window pre−draw hooks here ...
5     finally:
6         # This next line invokes the original flushWindow code
7         self.scottyFlushWindow()
```

Figure 3: Replacing the implementation of `NSWindow`'s `-flushWindow` method. The decorator on line 1 adds a new method, `scottyFlushWindow`, to the `NSWindow` class's method table, containing the code in lines 2–7. It then swaps the two implementations, such that when `scottyFlushWindow` is later called (line 7), it invokes the original Cocoa implementation.

Dynamic Code Support Runtime software development fundamentally relies on being able to dynamically load code into the program space of an existing program and on being able to change the implementation of its objects. First, we automate this code loading process using a plugin model. As described in the next section, a programmer simply writes a plugin. Loading it is handled automatically by our toolkit.

Second, we need to be able to override and overload method implementations in existing classes. Even in languages such as Objective-C that provide explicit support for doing so, this support may be cumbersome to use. We provide explicit support to simplify this process. The example in Figure 3 is used to implement some of the window hooks described earlier. Beyond Objective-C, we have implemented proof-of-concept versions of such method-swapping (also called method swizzling) that work on public methods in Java and on their equivalent in Javascript.

Object Proxies When adapting existing user interfaces it is common to need to modify a single instance object. For example, one might wish to change what happens when the user hovers over one particular button but not others. Object proxies provide a way of performing such an operation.

An object proxy is a metaclass that allows a developer to override, overload, or add new methods to a particular instance object. It uses an existing instance object to dynamically create a subclass of that instance's class. Any methods in the resulting proxy class invoke their equivalent in the proxied object, yielding its result if any. In this way, a developer can effectively replace a method's implementation at the object level rather than the class level. Object proxies thus permit the runtime wrapping of existing program objects.

While object proxies rely on the capabilities of the underlying object model and are used, for example, in some distributed environments, we use them to transform behavior

rather than to merely distribute it. To the best of our knowledge their use in the context of user interface toolkits is novel.

Code Inspection For some modifications, such as the Scribbler example below, it may be sufficient to be familiar with the programming toolkit used (*e. g.* Cocoa on Mac OS X, Swing for Java, jQuery or other Javascipt frameworks for Web applications) and the toolkit's inherent design patterns. For deeper modifications, however, a strong understanding of the toolkit is not sufficient. A third-party programmer must be able to discover certain aspects of the organization of the underlying code. For example, to modify the behavior of a particular button, it is necessary to know what method is invoked when it is clicked. While DOM inspection tools exist in some browsers, *e. g.* Safari and Firefox, to help debug web applications, they are not found in traditional toolkits.

To aid this discovery process, we provide several run-time inspection tools: a *hierarchy browser* to explore the application's window and widget hierarchy (Figure 7); an *object inspector* to list the class and methods of an arbitrary object; a *widget picker* that maps a click in the interface to the code object backing the clicked widget; and an *interactive interpreter* (Python) that runs inside the program, allowing interactive probing and inspection of the application, much in the same way as the original Smalltalk environment supported code inspection [13].

Together, these tools help decrease the complexity of making sense of an existing application. To the extent possible, our goal is to reduce the task of writing a third-party modification to that of writing a small program in the underlying programming environment.

Summary

The six abstractions described above can be grouped into two classes: those that let third-party developers create modifications to existing programs without deep understanding of their internal structures, and those that help them to make sense of and deeply integrate with the underlying implementation of the program. Event funnels, window hooks, and glass sheets fall into the former category, permitting developers to "bolt on" new behaviors. Dynamic code support, object proxies, and code inspection tools fall into the latter category, helping developers to figure out what parts of an existing program need to be modified and providing ways to do so in minimally-invasive ways.

THE SCOTTY DEMONSTRATOR

Scotty[5] is our reference implementation of runtime toolkit overloading. Its goal is to demonstrate the viability and utility of this approach in creating deep modifications to software applications *without access to their source code*. Scotty adds a plugin architecture to existing, unmodified Cocoa applications on Mac OS X. Runtime developers can use this toolkit to write plugins that augment, enhance, or otherwise modify the application at runtime. End users can then use these plugins to transform their interaction with the original application.

[5]Scotty is named after the miracle-working engineer on Star Trek and is available from `insitu.lri.fr/Projects/Scotty`.

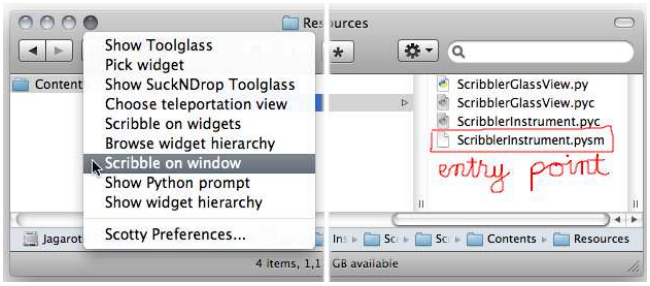


Figure 4: The Scribbler plugin running in the Mac OS Finder. The user activates the Scribbler instrument (left), and uses it to draw an annotation (right).

When a Cocoa application launches, Scotty bootstraps itself into the existing application environment. We describe how this bootstrap works in more detail in the Implementation section below. The first step of this bootstrap process is to confirm with the user that she wishes to activate Scotty, along with an option to remember that choice for the specified application. If the user chooses to activate Scotty, the available plugins will be looked up in the user's `Library` folder.

Scotty plugins are based on a model of instrumental interaction [4], where plugins define interactive instruments that augment the capabilities of the underlying application. A predefined plugin adds the Scotty button (the fourth button from the left in the titlebar in Figure 4) to the application windows. Clicking this button activates the current instrument, while pressing and holding it shows the Scotty menu, which lists the available instruments.

In the remainder of this section, we present a selection of plugins created using Scotty. We created these plugins out of real-world needs that have arisen in our own use, and also to show a breadth of the capabilities of these abstractions.

Creating a Plugin: Scribbler Plugins are simply folders that contain a description of the plugin, the plugin code, and any plugin resources. To give an idea of what it is like to create such a plugin, we describe how to create a simple *Scribbler* plugin that allows the user to draw annotations on a window. These annotations will be attached to the window, so that they move as the window moves. First, we create a folder, `Scribbler.instrument`, containing an `Info.plist` file. This file uses an XML structure to describe meta-data about the plugin, such as its name, associated code files, *etc*.

When a plugin loads, Scotty loads the code specified in the `Info.plist`. It then looks for any `Instrument` subclasses defined in the loaded code. For each new instrument (a plugin may define many), it invokes the `registerInstrument` class method, which, among other things, registers the instrument with the Scotty menu.

To create a Scribbler instrument, then, involves creating a Python file that defines an `Instrument` subclass, such as shown in Figure 5. When the user selects the instrument from the pulldown menu (Figure 4, left), Scotty queries the instrument to see if it needs a glass window (line 9). If it does, then it automatically creates one or reuses an exist-

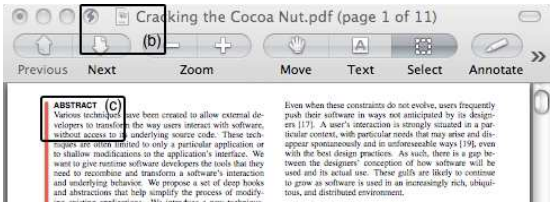
```
1  class ScribblerInstrument(Instrument):
2      @classmethod
3      def registerInstrument(cls, instrumentID):
4          # ... register for notifications if needed ...
5          super(ScribblerInstrument, cls).registerInstrument(...)
6      def __init__(self):
7          # ... setup scribbler data ...
8          self.stateMachine = self
9      def wantsGlassWindow(self):
10         return True
11     def newGlassViewForGlassWindow(self, glassWindow):
12         parent = glassWindow.parent()
13         view = ScribblerGlassView(parent.frame(), self)
14         return view
```

Figure 5: Scribbler instrument. The class method `registerInstrument` (lines 2–5) is called when the instrument plugin is first loaded to, *e. g.*, register for window creation notifications. In the constructor, the instrument registers itself as a state machine (line 8). This state machine (not shown) describes how the instrument translates input events into drawing on its glass view. Line 9 registers that the instrument will draw to a glass window, which will automatically be created. Line 14 returns a view (not shown) that will be added to that window to handle all drawing.

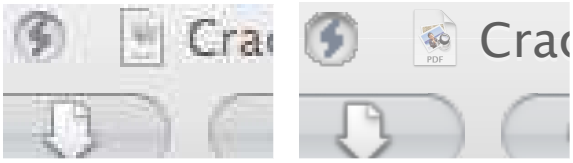
ing one. It then asks the instrument for a view to add to the glass window (lines 11–14). This view is responsible for any drawing that will be overlaid. Glass windows are themselves event funnels, redirecting all input events sent to the glass window to the associated instrument's state machine (set on line 8). The state machine describes which events are intercepted and how to handle each event. For brevity, we do not show the state machine itself. It describes a simple click-and-drag interaction to draw arbitrary shapes and remembers these shapes to redraw them when the window refreshes its content. The description of state machines uses a technique inspired by the Swing States library for Java [2].

Teleporting Windows to a High-Resolution Wall Display
When the user activates the window teleportation instrument, clicking on the Scotty button toggles a window's teleportation state (Figure 6). When a teleported window repaints, it must send a raster or vector representation of its content to a remote host, depending on which mode is active. Sending a raster representation is easily achieved by accessing the backing of the window. But in order to take advantage of the size and resolution of a wall display, we need a vector representation of the teleported window.

A vector representation describes a window's objects such as fonts, lines, primitive shapes, and more complex bezier paths in a resolution-independent way. SubArctic's graphics object wrapping approach works well in Java, where all drawing goes through an instance of a `Graphics` object that defines how each graphical object is drawn to the screen. On Mac OS X, however, there are several different kinds of drawing contexts: high-level Cocoa drawing contexts that use an object-oriented API, low-level Carbon contexts that use a C API, and OpenGL contexts. Because toolkit overloading operates on the object-oriented runtime layer, it cannot overload the lower-level C drawing operations; it only has access to the Cocoa-based drawing contexts. As such, we cannot easily



(a) Teleporting Preview, the Mac OS PDF viewer



(b) Close-up of the title bar in raster and vector mode

(c) Close-up of some text in raster and vector mode

Figure 6: (a) Teleporting the Preview PDF viewer. Enlarged detail regions are typeset directly from their window backings. (b) Bitmap data is scaled up as chunky pixels, but the icon remains crisp because it is backed by a higher-resolution image. (b) Rasterized text that would otherwise be difficult to read remains clear.

create a wrapped version of the drawing contexts that saves the vector data of objects drawn to the screen. Instead, we need another approach.

Fortunately, Cocoa includes printing hooks that access the lower-level vector representation of the contents of a window suitable for printing.[6] A programmer can thus use Scotty's window repaint notification to trigger the appropriate calls to this printing system, yielding a PDF representation of the window that can then be sent to a remote client (Figure 6).

This example illustrates two of the challenges involved in runtime software development. First, the programmer must be able to make sense of an existing program or, in this case, toolkit. For many kinds of modifications, as in this case, it is sufficient to know the toolkit used to build the application.

Second, access to the underlying program objects may sometimes be insufficient: Pragmatic choices made at design time or during the evolution of the software may restrict the operations that can be readily performed. As a result, it is sometimes easier to operate at a lower level of abstraction. For example, to transmit only the raster representations of windows or even widgets, surface representation-based approaches may be more appropriate. Indeed, when teleporting raster representations, Scotty uses a similar approach by accessing the window backings. When deeper access to the underlying program components is necessary, however, those methods fall short, and higher-level access is required.

[6] In practice, the printing system is much slower than drawing to the screen, but we have still been able to maintain interactive local frame rates on a 5-year-old 2 GHz Intel Core Duo laptop.

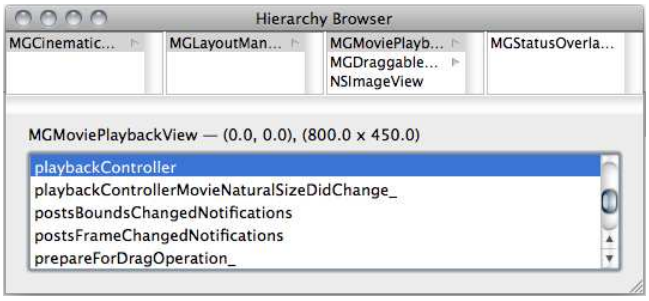


Figure 7: The hierarchy browser revealing the structure of the QuickTime Player window on the top and the methods available to the selected view below. Selected views in the browser are highlighted in the running application (not shown).

Adding Subtitles to DRM-encumbered Movies We want to integrate external subtitles into the standard Mac OS Quicktime Player, which is capable of playing the DRM-locked videos purchased from the iTunes Store. In order to properly handle the subtitles, the text must be properly laid out on the screen, even if the user resizes the window. The subtitles must also stay synchronized with the video, even if the user skips around in the video.

A solution to this problem must solve two challenges: to display subtitles on top of Quicktime Player's video playback view, and to keep them synchronized with the video playback as the user plays, pauses, and scrubs through the video. A programmer will thus not only have to figure out how to solve the tasks of his problem domain, *i. e.*, fetching the subtitles, but also how to integrate with an unfamiliar codebase.

The first part of this problem is easy to solve by attaching a glass window to the video playback widget. Glass windows automatically register for resize notifications on their parent. By tracking these notifications, the glass window will automatically resize with the parent, keeping its contents (in this case, the subtitles) in a consistent location on the screen, even as the user resizes the window or enters and exits fullscreen mode. The programmer need only determine to which widget to attach the glass window, in this case using the widget hierarchy browser (Figure 7).

The second challenge is in keeping the subtitles synchronized with the video playback. Using the hierarchy browser, the developer identifies the method of the playback view (`MGMoviePlaybackView`) that gives access to the video controller object (`MGMoviePlaybackController`) . With access to the underlying video controller, the developer can directly handle changes to the video playback properties, regardless of whether the user presses a button on the screen, uses a keyboard shortcut, or a remote control to play, pause, or skip around in the video. The key is that instead of reacting to input events, we interact with their associated logical operations at the controller level, *e. g.* play or skip.

Adding New Toolbar Commands Most video players offer the ability to make playback windows float on top, with the notable exception of the Quicktime Player in recent versions of Mac OS. For this and other applications, it would be useful to be able to add a new window button that toggles this floating property. More generally, we want to be able to dynamically create new buttons for windows and toolbars and even to redefine the behaviors of existing buttons.

Scotty integrates an interactive Python interpreter into existing Cocoa applications. Using this interpreter, the runtime developer can interactively enter code, define new classes and even add or replace methods on existing classes. She can interactively develop and test a new feature, such as making a window float above others using an `NSWindow`'s `-setLevel:` method, a standard method in the Cocoa API, but one that few applications make available in their interface. Once refined, she can add a new window button for that function. She can first test the button by writing the code that creates it interactively in the code window, and then integrate it into the plugin code. Once part of the plugin, that button will be added to all new windows with title bars via a window creation hook. In some sense, this is akin to creating Buttons [17], but within the application.

In addition to adding new buttons to a window, we can also redefine button actions. For example, when sending a message in Apple's Mail program, we would like to inspect the contents of the message to verify that the user has not forgotten to attach a referenced document. We do not want to merely intercept pressing the Send button, but also its equivalent interactions such as a menu item or keyboard shortcut.

Cocoa uses a target-action pattern for callbacks. In a well-written program, the action associated with a button is written in a generic action handler that is shared across each of these different triggers. The runtime developer therefore first needs to find the callback invoked when the button is pressed. She then needs to perform her domain-specific modification to access the email body and attachments.

To find the callback associated with the send button, she could use the hierarchy browser to find the button. Alternatively, she could activate the widget picker instrument, which maps a click to a widget, giving access to its target and action. Thus, the programmer can simply click on the Send button using the widget picker instrument to reveal that she needs to modify the `-send:` method of the `MailDocumentEditor` class. Further exploration of the methods listed in this class reveals how to access the message body and attachments in the associated message. Using the interactive interpreter, the programmer can confirm that these methods do indeed yield the anticipated results. After refining her code, she then saves the modifications so that they will be applied in subsequent program invocations.

SCOTTY IMPLEMENTATION

Runtime toolkit overloading relies on two primary elements: the ability to dynamically replace method implementations at the object-oriented dispatch level and a well-defined object-oriented toolkit built on that object system. Object-oriented programming languages such as Python, Java, Smalltalk, Objective-C, Javascript, and even C++[7] all use dynamic dispatch in method invocation. This dynamic dispatch is governed by

[7] when the `virtual` keyword is used

a small runtime component that maps the name of a method to the code that implements it, usually based on the type of the object invoked. It is this runtime component that ensures that the code `widget.trigger(sender)` will invoke the appropriate `trigger` code regardless of whether the widget is a button, a menu item, or a text field. By dynamically redefining these mappings, we can change the behavior of a class.

We implemented Scotty on Mac OS X using Cocoa, which implies the use of Objective-C. One feature of this language is that it provides access to this runtime component in the language. In other languages, the difficulty of this task depends on how well-defined the runtime component is, whether it is standardized across compilers, and whether the language provides hooks to access it. We have successfully implemented method swizzling in both Java (for Swing) and in Javascript (for web applications). In Java we use the introspection API to change the implementation of any public method. In Javascript, we use the delegation model to achieve the same effect.

Scotty is implemented in Python using the Python/Objective-C bridge, PyObjC[8]. Although one is interpreted and the other is compiled, Python and Objective-C have a very compatible runtime object system. PyObjC provides a two-way dynamic bridge, allowing classes in either language to subclass those defined in the other. As such, we can take advantage of Python's introspective capabilities to interact with the Objective-C environment.

Bootstrapping

In order to use our runtime modifications in existing programs, we need a way to be able to *dynamically* modify the object-oriented method and class tables. We need to be able to run our modifications in the same program space as an arbitrary Cocoa application. We use an old feature of the Cocoa runtime, called Input Managers, that enables 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 application.[9]

A similar result can be achieved in other environments. One solution is to take advantage of the dynamic loading of shared libraries, an approach used by Besacier and Vernier in the Windows environment [6]. In the web environment, Greasemonkey, among others, uses the extension capabilities of modern browsers such as Firefox and Safari to inject code into any loaded page. We have successfully used a modified class loader in Java and browser extensions in Javascript to demonstrate that bootstrapping works for both Swing applications and web applications.

Once loaded inside of the program space, Scotty first creates a `ScottyController`, a singleton manager that governs the loading of Scotty plugins, preferences, *etc*. This manager then grafts itself into the Cocoa environment, overloading the implementations for `NSWindow`'s `-flushWindow` method (Figure 3) and the windows' initializers (not shown). These methods provide access to Cocoa's window creation and rendering operations. It also taps into the event dispatch system to intercept UI events sent to the application but not associated with a particular window, such as mouse motion events over the desktop when the application is active.

Obviously, replacing methods of an existing application is dangerous. The replacement method will be called in place of the original one, so it must not break any of the implicit or explicit assumptions made by the application designers. It is, therefore, a brittle operation that requires care and defensive coding practices. To reduce the risk of breaking applications, we minimize the number of methods that are replaced and we provide developers with higher-level abstractions and wrappers that are safer than directly replacing methods.

DISCUSSION

The examples we have developed with Scotty illustrate how runtime toolkit overloading can help a runtime developer to alter the behavior and interface of existing applications, building on the interface and functionality that are already present in the application. The Scribbler example shows how to add a new capability without deep understanding of the toolkit. The teleportation example shows how to make deeper modifications on the toolkit level. In both cases, the user need only be familiar with the standard toolkit API and the Scotty abstractions. Sometimes, however, deeper integration with a particular application is necessary, as shown in the *ad hoc* modifications to Quicktime Player and Mail.

In both the Quicktime and Mail examples, the developer must figure out what parts of code to interact with without having a published API to help. In both cases, the fact that the user interface provides a human-interpretable interface to the underlying program objects gives the programmer a tractable starting point. Using either the window hierarchy browser or the widget picker, she can get a handle on the program object associated with an arbitrary widget. She can then browse the list of that object's methods similarly to an (undocumented) API summary listing. In this way, when a developer does need to deeply integrate with the core of an application, Scotty helps guide that search based on the externally visible parts of the software, that is, the interface.

The various examples show a cross-section of the kinds of modifications that runtime toolkit overloading makes possible. Augmented surface representation approaches could potentially support some of these modifications. Façade, for example, could modify a window's position in the window hierarchy to have it float over all other windows. Sikuli's visual scripting interface [24] could be used to create automated tasks. For window teleportation, however, surface representation methods do not have access to the underlying structures to send an alternative vector representation.

Interacting with the underlying application logic is simply not possible with surface representation approaches. Checking for missing Mail attachments and properly synchronizing subtitles in the Quicktime video player requires inspecting the underlying application models. While such deep integration may be more challenging than surface modifications, it enables modifications that were previously impossible.

[8] PyObjC: `pyobjc.sourceforge.net`, 20 April 2011.

[9] SIMBL, the Smart Input Manager Bundle Loader, uses this same process. `www.culater.net/software/SIMBL/SIMBL.php`, 20 April 2011.

Runtime Program Modification

More generally, approaches to runtime program modification can be characterized along two dimensions: *depth of integration* with the underlying application and *generality* of the approach. Depth of integration refers to how much access the technique provides to a program's underlying objects. The generality of the approach describes how well the technique applies to all applications or to specific ones. The work presented in this paper offers both deep integration and generality across a toolkit.

Approaches such as scripting and extension interfaces can be deeply integrated with the application and can offer a rich degree of modification. These interfaces allow a third-party programmer to create rich modifications to the underlying behavior of the program, but with the risk that those modifications could cause instability, data corruption, or unexpected interactions with the program. They are powerful, but with this power comes a higher risk. To mitigate this risk, the abstractions we have used reduce the surface area of the underlying application requiring modification.

At the other end of the spectrum are shallow integration techniques, including surface representation approaches and their augmented variants. Their lower integration footprint helps make them less fragile, but they are still not without risk. For example, if a bug in a surface level modification sends multiple mouse release events without an intervening mouse press event, the application may crash. Furthermore, these approaches often cannot completely graft a new behavior into an existing application or toolkit. They may be able to create close approximations, but they lack the tight integration necessary to handle edge cases, and they cannot handle cases that require deeper semantic understanding of program objects.

Surface level approaches, however, are typically more general across applications built with different toolkits. Façade can handle any X Windows applications, regardless of what toolkit it uses, because it integrates with the underlying windows on which those toolkits are built. Prefab can handle any widget it has been previously trained to recognize, regardless of whether it is running locally, inside of a remote VNC connection, a virtual machine, embedded in a Flash application, or just simply in a screenshot. As such, they can potentially offer more general, toolkit-independent solutions, but with a limited understanding of the underlying program.

Deciding which approach to use involves weighing the needs of the particular modification against the risk involved in modifying an application's behavior. One of the advantages of the abstractions we have developed is that they support a similar degree of input/output hijacking as existing surface representation approaches. Runtime program modification using our abstractions therefore faces a risk similar to surface-level approaches, with the important difference that it runs inside the application. This approach may therefore offer the best of both worlds when using a single toolkit.

FACILITATING RUNTIME SOFTWARE DEVELOPMENT

Runtime software development relies on the ability to examine and modify live objects in program space, and the ability for the runtime developer to make sense of those objects. The first of these relies on three technical abilities: to run arbitrary code in program space, to modify class and method tables at runtime, and to intercept and redirect input events before delivery. While our demonstrator is implemented in Cocoa, we have implemented proofs of concept in both Java and Javascript, demonstrating the generality of the approach.

In all three cases however, we rely on specific and sometimes obscure features of each environment. A much better approach would be for the UI toolkit itself to directly support the type of extensibility we have described. Window Hooks, Event Funnels, and Glass Sheets can easily be natively supported by new toolkits. Dynamic Code Support and Object Proxies require proper runtime support at the programming language level, but could be explicitly supported by the toolkit for its own objects and methods. The bootstrapping process could use the same approach as applications that support plugins, *i. e.* to load the content of a well-known directory at start-up.

Finally, Code Inspection is mostly a human issue: how to help a runtime developer make sense of someone else's program. Even with access to source code, the task is difficult. The inspection tools in Scotty go some way to helping make sense of the code, but could be improved. This task is simpler than general code comprehension because it only involves methods and objects connected to elements of the user interface. Since the user interface is intended to be human understandable, it should support the exploration process, as we have done with the simple widget picker.

CONCLUSIONS & FUTURE WORK

The goal of runtime software development is to support the extension and adaptation of existing software applications without access to their source code. While some tools exist, they are usually somewhat limited, and programming extensions and adaptations proves difficult.

We have presented *runtime toolkit overloading*, an approach that makes it possible to perform powerful adaptations on existing applications without modifying to their source code. Specifically, we have shown how Window Hooks, Event Funnels, Glass Sheets, Dynamic Code Loading, Object Proxies, and Code Inspection help to augment or replace behaviors of existing program objects. We have demonstrated Scotty, a prototype system that implements these abstractions in existing Cocoa applications in Mac OS X. We have shown how Scotty can be used to implemented a variety of modifications, including teleporting vector or raster representations of windows, adding subtitles to a video player, and dynamically adding or redefining toolbar buttons.

We have shown how to create and use runtime toolkit overloading in the Cocoa environment and have created proofs of concept in two other application environments: Java/Swing and web applications. We further argue that new toolkits can better support runtime development by directly incorporating these abstractions into their design.

In future work, we will study how this approach can better foster adaptation as a core design concept, facilitating the

development of applications that not only support users at altering their interaction and behavior, but also by incorporating runtime adpatation as an explicit design element.

ACKNOWLEDGEMENTS

This work was supported in part by the French ANR grant "iStar" (#2007-TLOG-009-03) and by the Digiteo/Région Île-de-France grant "WILD" (#2008-25D).

REFERENCES

1. J. Accot and S. Zhai. More than dotting the i's — foundations for crossing-based interfaces. In *CHI '02: Proceedings of the SIGCHI conference on Human factors in computing systems*, 73–80, New York, NY, USA, 2002. ACM.
2. C. Appert and M. Beaudouin-Lafon. SwingStates: adding state machines to Java and the Swing toolkit. *Softw. Pract. Exper.*, 38(11):1149–1182, 2008.
3. Apple Computer Inc. *AppleScript Language Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.
4. M. Beaudouin-Lafon. Instrumental interaction: an interaction model for designing post-wimp user interfaces. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, 446–453, New York, NY, USA, 2000. ACM.
5. J. M. A. Begole. *Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems*. PhD thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1998.
6. G. Besacier and F. Vernier. Toward user interface virtualization: legacy applications and innovative interaction systems. In *EICS '09: Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, 157–166, New York, NY, USA, 2009. ACM.
7. M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller. Automation and customization of rendered web pages. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, 163–172, New York, NY, USA, 2005. ACM Press.
8. A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch what I do: programming by demonstration*. MIT Press, 1993.
9. M. Dixon and J. Fogarty. Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*, 1525–1534, New York, NY, USA, 2010. ACM.
10. M. Dixon, D. Leventhal, and J. Fogarty. Content and hierarchy in pixel-based methods for reverse engineering interface structure. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, 969–978, New York, NY, USA, 2011. ACM.
11. W. K. Edwards, S. E. Hudson, J. Marinacci, R. Rodenstein, T. Rodriguez, and I. Smith. Systematic output modification in a 2d user interface toolkit. In *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology*, 151–158, New York, NY, USA, 1997. ACM.
12. W. K. Edwards, E. D. Mynatt, and K. Stockton. Providing access to graphical user interfaces—not graphical screens. In *Assets '94: Proceedings of the first annual ACM conference on Assistive technologies*, 47–54, New York, NY, USA, 1994. ACM.
13. A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1983.
14. G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 693–702, New York, NY, USA, 2002. ACM.
15. G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
16. W. E. Mackay. Triggers and barriers to customizing software. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, 153–160. ACM Press, 1991.
17. A. MacLean, K. Carter, L. Lövstrand, and T. Moran. User-tailorable systems: pressing the issues with buttons. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, 175–182, New York, NY, USA, 1990. ACM Press.
18. B. A. Nardi. *A small matter of programming: perspectives on end user computing*. MIT Press, 1993.
19. T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, Jan/Feb 1998.
20. M. Robinson. Design for unanticipated use... In *ECSCW '93: Proceedings of the third conference on European Conference on Computer-Supported Cooperative Work*, 187–202, Norwell, MA, USA, 1993. Kluwer Academic Publishers.
21. W. Stuerzlinger, O. Chapuis, D. Phillips, and N. Roussel. User interface façades: towards fully adaptable user interfaces. In *UIST '06: Proceedings of the 19th annual ACM symposium on User interface software and technology*, 309–318, New York, NY, USA, 2006. ACM.
22. L. A. Suchman. *Plans and Situated Actions: The Problem of Human-Machine Communication*. Cambridge University Press, December 1987.
23. D. S. Tan, B. Meyers, and M. Czerwinski. Wincuts: manipulating arbitrary window regions for more effective use of screen space. In *CHI '04 extended abstracts on Human factors in computing systems*, 1525–1528, New York, NY, USA, 2004. ACM.
24. T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: using gui screenshots for search and automation. In *UIST '09: Proceedings of the 22nd annual ACM symposium on User interface software and technology*, 183–192, New York, NY, USA, 2009. ACM.