Grab 'n' Drop: User Configurable Toolglasses

James R. Eagan(✉)

LTCI, Télécom ParisTech, Université Paris-Saclay, Paris, France
james.eagan@telecom-paristech.fr

Abstract. We introduce the *grab 'n' drop toolglass*, an extension of the toolglass bi-manual interaction technique. It enables users to create and configure their own toolglasses from existing user interfaces that were not designed for toolglasses. Users compose their own toolglass interactions at runtime from an application's user interface elements, bringing interaction closer to the objects of interest in a workspace. Through a proof-of-concept implementation for Mac OS X, we show how grab 'n' drop capabilities could be added to existing applications at the toolkit level, without modifying application source code or UI design. Finally, we evaluate the power and flexibility of this approach by applying it to a variety of applications. We further identify limitations and risks associated with this approach and propose changes to existing toolkits to foster such user-reconfigurable interaction.

Keywords: User interfaces · Toolglasses · Instrumental interaction · Polymorphism

1 Introduction

Toolglasses [4] are a bi-manual interaction technique in which users click through movable controls to apply their operations to the target below. They have been shown to be useful in a variety of contexts, from drawing applications [4] to debugging tools [13] to editing colored Petri nets [2].

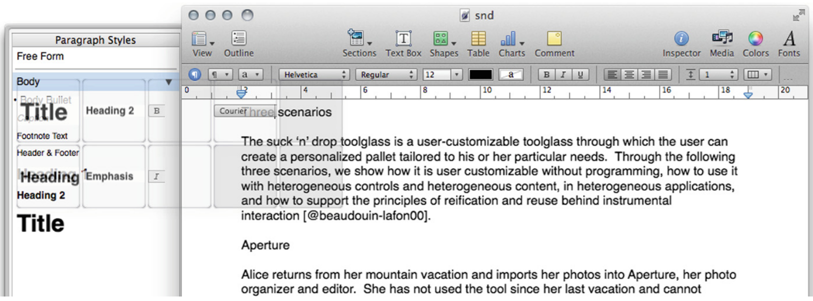
In the nearly 25 years since toolglasses were initially proposed, multiple pointing devices have become increasingly common. While still not the norm, many laptop users, for example, work with an external mouse, effectively providing a second pointing device in addition to the laptop's integrated pointer.

Despite their utility, few applications provide support for toolglasses. In this article, we show how existing user interface (UI) toolkits could be updated to provide support for user-configurable toolglasses, without requiring application programmers to update their code. Users can then assemble and configure their own toolglasses from existing applications. We introduce the *grab 'n' drop toolglass*, a proof-of-concept implementation for Mac OS to explore what existing toolkit features help make this kind reconfiguration feasible, where they are insufficient, and how toolkits could be modified to better support such reconfiguration. The running toolglass prototype probes where, in real applications,

R. Bernhaupt et al. (Eds.): INTERACT 2017, Part III, LNCS 10515, pp. 315–334, 2017.
DOI: 10.1007/978-3-319-67687-6_21

this approach breaks down. While this implementation is in the context of Mac OS, it uses principles that are generalizable across most common UI toolkits. We further explore several ways that users might be able to extend such reconfigurable interfaces and discuss design limitations in current toolkits that limit the kinds of extension possible.

The contribution of this work is thus (1) an interaction technique for assembling user-created toolglasses from existing interface elements in existing applications and (2) a technical contribution of how existing toolkits could be updated to better support such toolglasses. We evaluate the grab 'n' drop toolglass not from a usability point of view but in terms of the technical feasibility of transforming interaction by users.



three scenarios

The suck 'n' drop toolglass is a user-customizable toolglass through which the user can create a personalized pallet tailored to his or her particular needs. Through the following three scenarios, we show how it is user customizable without programming, how to use it with heterogeneous controls and heterogeneous content, in heterogeneous applications, and how to support the principles of reification and reuse behind instrumental interaction [@beaudouin-lafon00].

Aperture

Alice returns from her mountain vacation and imports her photos into Aperture, her photo organizer and editor. She has not used the tool since her last vacation and cannot

Fig. 1. The grab 'n' drop toolglass in Pages. The toolglass, floating on the top left, contains three empty buckets on its bottom, right. The remaining buckets contain the extracted actions from various widgets visible in the toolbar and styles list on the left. The user has just grabbed the "Heading 1" style.

Initially, a grab 'n' drop toolglass provides a collection of empty buckets (Fig. 1). When the user clicks through an empty bucket onto a widget, the toolglass stores the action associated with that widget (grab). When the user subsequently clicks through the toolglass, it applies that action to the object under the toolglass (drop). As such, a user can add and configure toolglass interactions, even in applications not designed for such interaction.

Furthermore, these toolglasses can be polymorphic. For example, a user may reconfigure a grab 'n' drop toolglass created in one application to perform analogous operations in another. A polymorphic operation is a logical operation that is realized differently depending on some combination of the control, the target object, and its context. For example, applying a heading style to some text in a word processor might set a named style to the text; in an HTML or LaTeX editor, it might wrap it in markup; in an email message it might set it to bold and change its size. A single logical operation—make this text a header—may be implemented using different underlying functional operations depending on the context.

2 Related Work

The grab ‘n’ drop toolglass builds on toolglasses, certainly, but also on work in runtime, third-party program modification, on end-user customization, and on user-interface toolkits and interactions.

Toolglasses and magic lenses [4] were first proposed by Bier *et al.* in 1993 as a general-purpose interaction technique, a new style of widget that programmers could add to their toolbox. Various extensions have been proposed, including toolglasses for debugging [13] and visualization [11]. Moreover, they act as a strong complement to other tools, such as floating palettes and marking menus [20].

Perhaps the closest in spirit to our proposed technique is the style toolglass in Beaudouin-Lafon and Mackay’s CPN2000 [3]. Initially, empty items in the toolglass act as a style picker, extracting attributes such as color and thickness. They then become style droppers, applying those attributes to target objects. We extend CPN2000’s notion of picking (grabbing) and dropping styles beyond an application’s domain objects to its interface elements.

Other approaches let users modify a program’s interface: Tan *et al.*’s WinCuts [27] let users trim away irrelevant portions of windows to their context. Hutchings and Stasko’s window snipping extends this notion to provide for live interaction with trimmed windows [14]. Pushing the farthest, Stuerzlinger *et al.*’s user interface Façades [26] treat the interface itself as a modifiable object, letting the user recompose those elements to create new palettes or even replace tools with functional equivalents. While they do enable users to convert entire tool pallets and toolbars into toolglasses, they does not describe support for more general composition from more primitive widgets. We generalize this approach to individual widgets and extend it beyond command-then-object interactions.

Program Modification. Various approaches have been taken to enabling third-party modifications to software. Cypher *et al.* edit a good overview of such modifications for the web [6], where a strong mash-up culture has evolved. On the desktop, tools such as Dixon and Fogarty’s Prefab [8] offer the possibility to “make every application open source” by identifying the user interface components in the pixels drawn to the screen. Such approaches are surprisingly powerful, despite only having access to rendered pixels and user input events. Other approaches involve varying degrees of access to or integration with source code. Eagan *et al.* provide an overview of such techniques [10]. Each of these approaches aims to provide developers with the means to add new functionality or behaviors to existing software.

End-User Customization. Non-programmer end-users have a long history of customizing their software to suit their own particular needs [19]. With sufficient scaffolding and motivation, they can even go as far as to program [9,22], and even construct sharing communities [12,17], with different users assuming different roles to encourage and foster such customization [12].

Our work draws inspiration from these approaches. In this context, our goal is to enable users to compose new interactions from existing program functionality without requiring them to program or manipulate complex configuration dialogs. As such, we draw inspiration from programming by demonstration [7]: a user simply clicks through interface elements to construct a personal toolglass. Grab 'n' Drop toolglass do not, however, attempt to infer intent from user's actions. As such, they could best be described as employing configuration by demonstration rather than programming by demonstration.

Finally, Ponsard and McGrenere propose Anchored Customization [23], which modifies the preferences panels of existing applications to enable users to access preference settings directly from the user interface elements involved. This approach is similar in two primary ways: in both cases, the proposed technique brings the user's interaction and attention to the relevant widgets; and both techniques were grafted into existing applications by means of a hack that is independent of the technique itself.

3 Four Scenarios

The grab 'n' drop toolglass is a user-customizable toolglass through which the user can create a personalized pallet tailored to his or her particular needs. Through the following scenarios, we show how it is user customizable without programming, how to use it with heterogeneous controls and heterogeneous content, in heterogeneous applications, and how to support the principles of reification and reuse behind instrumental interaction [2].

3.1 Aperture

Alice returns from her mountain vacation and imports her photos into Aperture, her photo organizer and editor. She has not used the tool since her last vacation and cannot remember the keyboard shortcuts when operating in full-screen mode, so she creates a new grab 'n' drop toolglass in standard, windowed mode. Into four empty buckets, she clicks through the toolbar buttons for the straighten, crop, redeye, and retouch commands. She then enters full-screen mode, which hides all controls but the cursor and her toolglass, and expands the photo to fit to the undecorated screen. She clicks through the crop toolglass to re-compose the image, perhaps refining the crop by moving the toolglass out of the way. She then clicks through the red-eye reduction tool to hide the effects of the flash.

3.2 Styles in Pages

Bernard and Charlie are collaborating on an article for Interact, and Charlie has composed an unformatted first-draft of the article. In order to apply the correct styles to the article in Pages, Bernard must first select the relevant text or paragraph, then select the appropriate style in a list on the side of the screen. Systematically applying these styles thus involves repeatedly scrolling and precisely pointing in the document, then precisely pointing in the list on the side, alternating attention and focus between the content and the styles palette (Fig. 1).

Bernard creates a new grab ‘n’ drop toolglass and clicks through empty buckets to add the section, subsection, and other styles from the styles panel to the toolglass. He then closes the styles panel and clicks through to the bold and italic toolbar buttons to add them to the toolglass. He can then use this toolglass to systematically apply the appropriate styles to the document by scrolling and clicking through the toolglass within the document view, maintaining his attention on the text content without the back-and-forth from before.

3.3 Styles in Mail

Some time later, Bernard is working in Mail and wants to style a rich-text email message he is composing. He loads the toolglass that he created in Pages. Although Mail does support rich text, it does not use the same styles functionality as Pages. As a result, Bernard must first re-define each of the grab ‘n’ drop buckets in the toolglass by clicking through on the relevant controls in Mail. He holds the shift key to combine (or stack) multiple actions and clicks through the Helvetica font pull-down button, the bold button, and the 12-point font selector. Once defined, clicking on a particular style in the toolglass in Mail will apply a similar styling to text in a mail message as it would to text in his word processor.

3.4 Lab Notebook

Denise is documenting a lab procedure in an email. In the control software for a piece of lab equipment, she creates a new toolglass for the procedure and clicks through each of the relevant controls to extract its associated action. She then switches to her email message and command-clicks the controls in the toolglass to drop their stored screenshot as a rich-text attachment, integrating screenshots of the configured controls into the text of the procedure.

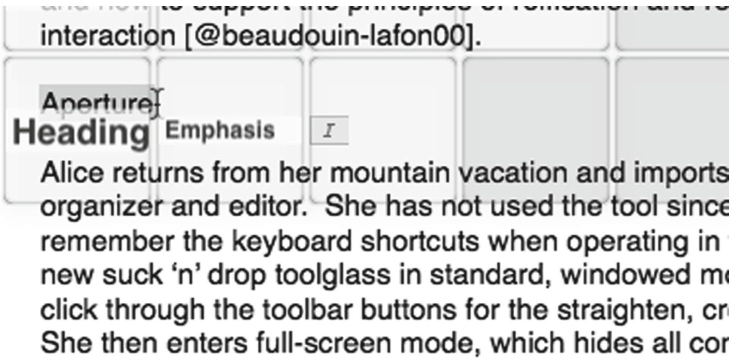
Some time later, Eric reads the procedure and command-clicks through an empty bucket onto the image of one of the controls embedded in the document in order to grab it into the toolglass. He then command-clicks through the bucket onto the relevant control to apply the state described in the image to the underlying control.

4 Grab ‘n’ Drop

The grab ‘n’ drop toolglass is based on traditional toolglasses, initially proposed nearly 25 years ago [4]. As with traditional toolglasses, it uses two pointing devices, such as an integrated trackpad in a laptop and a wireless mouse. The left (non-dominant) hand controls the position of the toolglass, while the right (dominant) hand controls the position of the pointer.

With the grab ‘n’ drop toolglass, the user can compose new toolglasses from the existing functionality exposed through the application’s existing interface, even in applications that do not currently support toolglass-based interaction. (Our prototype implementation is built on Mac OS X and integrates with arbitrary Cocoa applications using the Scotty toolkit [10].)

Initially, the toolglass is populated with empty controls, or buckets (Fig. 1). These buckets can be filled by positioning an empty bucket on top of a clickable control, such as a button, segmented control, list item, or toolbar item. When the user clicks through the empty bucket, the toolglass inspects the underlying control to extract its associated action. (We describe this process in more detail in the implementation section.) Once the bucket has “grabbed” the associated action, it draws itself with a copy of the control’s on-screen representation.



interaction [@beaudouin-lafon00].

Aperture

Alice returns from her mountain vacation and imports
organizer and editor. She has not used the tool since
remember the keyboard shortcuts when operating in
new suck ‘n’ drop toolglass in standard, windowed m
click through the toolbar buttons for the straighten, cr
She then enters full-screen mode, which hides all cor

Fig. 2. Dropping an action in Pages. The user has just selected the word “Aperture” through the “Heading 2” bucket of the toolglass. As soon as she releases the mouse, the style will be applied.

By selecting a collection of desired controls, a user can create his or her own custom toolglass. Furthermore, by storing the control’s target and associated action, the source control itself need not continue to exist. If, for example, the user populates one of the buckets with a control in a palette and later closes the palette, the bucket will continue to function.

When the user clicks on a filled bucket, the toolglass passes the click through to whatever window is underneath (Fig. 2). The press, any subsequent drag events, and the release are all passed through the toolglass to the underlying window. Each bucket can either trigger the associated action either just before relaying the user’s initial press to the region below the toolglass or just after the user releases the mouse. Activating the control before passing-through mouse events is useful for controls that activate modes (command-then-target), whereas the latter option is appropriate for controls that operate on a selection (target-then-command). If the user drags beyond an individual toolglass item, such as when highlighting a long sentence, the command used is that that was under the initial mouse press.

Toolglasses can be saved and restored in future invocations of the application, allowing for use for both one-off tasks and for co-adaptive behaviors [18]. In this way, a user can propose new interactions with an existing interface, making the application better fit his or her particular needs. Furthermore, he or she is likely to adapt his or her behavior as the interface itself now supports new kinds of interaction.

4.1 Polymorphic Toolglasses

Beyond inherently polymorphic commands, such as copy and paste, the grab 'n' drop toolglass supports polymorphism across applications by letting the user re-configure a toolglass for use in another application.

A toolglass loaded from another application will keep the same buckets, with each bucket showing the control's icon from the application in which it was initially created. Because each application is different, however, the controls themselves may or may not exist. As such, the bucket invalidates its references to the underlying control actions when loaded into a new application and draws itself in an inactive state. When the user attempts to use such an inactive bucket in a new application for the first time, the application prompts to reconfigure it for the new application by grabbing a new control. Holding the shift key appends additional controls.

In this way, the user can create a toolglass whose functionality adapts itself to the particular application in question. For example, applying a header style from the toolglass in a word processor might change a style by applying its word-processing concept of a style. In an HTML editor, it might trigger a command to wrap that text in a header tag. In a Mail client, it might make the text bold and change its size. In this way, a single bucket with a single logical function may trigger different underlying functionalities depending on the particular program.

4.2 Command Syntax

As the name suggests, there are two primary interactions associated with the grab 'n' drop toolglass: *grab* and *drop*. As described above, the grab operation extracts the associated action (and state, if applicable) of a control into the toolglass. The drop operation applies that action (including the associated state, if any) to the subject of the interaction with the underlying window below the toolglass. There is additionally a toggle to configure the *drop* operation to apply its associated action *before* or *after* the user interaction has occurred.

In the current proof-of-concept implementation, we use modifier keys at the time of the grab operation to regulate this mode. We plan to replace this interaction with something more intuitive in a future prototype. A modifier-less click applies the operation after passing through user interactions to the underlying window, as for a control that operates on a selection. Holding down the option key during a grab operation will cause the drop operation to apply before passing along the interactions, as suitable for a mode-switching control such as a control pallet item.

Normally, once a command has been grabbed into a bucket, subsequent clicks on that bucket are treated as drops. A user can combine multiple commands, however, by holding the shift key to grab additional controls into the toolglass. When a bucket contains multiple controls, a drop operation applies each action in the order in which they were grabbed.

Finally, holding the command-key during a drop causes the toolglass to drop the captured human-readable image of the control instead of applying the action. This image contains a serialization of the stored data in its meta-data, such that the image actually contains the same machine-readable data stored in the toolglass. A command-drop on a control, on the other hand, reconfigures the state of the control to reflect that stored in the toolglass. Holding the command-key during a grab operation causes the operation to look for such meta-data in an image under the mouse instead of looking for a widget at the click location.

4.3 Live Screenshots

The typical use of grab 'n' drop toolglasses is to compose a personal toolglass out of existing widget components, effectively treating the program's interface as malleable and re-composable. We further extended the toolglass to support treating the live interface as an embeddable object in a document. As such, the user can drop the captured screenshot of the widget in a bucket into a document, as in the lab procedure scenario described above. Indeed this scenario itself comes from Klokmose and Zander's work with scientists on laboratory notebooks [16].

When the user command-drops a bucket, the toolglass passes the click through to the underlying window, then inserts a screenshot of the widget as if pasted from the clipboard. In the meta-data of the image, the toolglass writes the same information that would be serialized for subsequent program invocations (described in more detail in the following section). As such, the dropped image contains both a human-readable representation of the associated widget, but also a machine-readable description sufficient to re-connect the image to the associated widget, similarly to how Kato *et al.* encode robot pose data in images embedded in source code [15].

Because all of the machine-readable information necessary to link the image to the widget is stored in the meta data of the image, any existing document or image editor that preserves the embedded meta-data can be used to edit that document. Re-establishing the link requires only that the toolglass be able to map a click to the image, which is currently implemented using the Mac OS X Cocoa rich text editing APIs.

4.4 Serialization

Once the user has created a personal toolglass, such as the styles toolglass, she can save that toolglass for later. Saving a toolglass causes each bucket's screenshot, associated action, and any state information about the widget to be serialized and stored in a file on disk. On a subsequent invocation of the application, the user can then load the toolglass to restore these links. Because the application was not designed for this kind of interaction in mind, re-establishing this link can be potentially fragile and does not work in all cases. We describe the details of this process in the Implementation section, and focus on specific edge cases in the Discussion section, below.

Serialization for Other Applications. In some ways, loading a saved toolglass in another application is more straight-forward. The grab 'n' drop toolglass relies on references to the underlying callback controllers of an application. With the exception of standard, toolkit-level actions, any two arbitrary applications are unlikely to use the same callbacks, even for logically-equivalent commands. As such, it is generally not possible to re-link a toolglass bucket from one application to a new application. For this reason, buckets loaded from another application are initially disabled. Effectively, these buckets are like empty buckets: a click through the bucket will grab the action off the target widget.

Unlike empty buckets, however, the bucket will still contain the original widget's screenshot and will keep that screenshot even after being re-linked in the new application. We chose not to replace the screenshot so as to maintain a more consistent representation across applications. For example, a styles toolglass is a styles toolglass, regardless of the particular operations necessary to realize it in the host application.

5 Implementation

We have created a proof-of-concept implementation of the grab 'n' drop toolglass as a Scotty [10] plugin for Mac OS X Cocoa applications. Eagan *et al.*'s Scotty uses runtime toolkit overloading to enable the modification of third-party applications without access to their source code[1]. Scotty plugins run inside the host program and thus have access to its internal classes and objects, including documents, views, and their controllers.

While this proof-of-concept implementation is effectively a Mac OS X hack using Scotty, neither the concept of the grab 'n' drop toolglass nor the overall implementation approach is dependent on the Mac. Rather, this approach enabled us to effectively extend the Cocoa toolkit to provide a set of generally reusable components that could be provided by other toolkits such as Java Swing or Windows .Net using analogous capabilities in those toolkits.

5.1 Requirements

There are four principal challenges in implementing grab 'n' drop toolglasses:

- Extracting and invoking a widget's associated action.
- Handling state and other side-effects.
- Managing object-then-action and action-then-object command syntaxes.

[1] We will make the source code available at `code.eagan.me`.

– Serializing these extracted data for subsequent program invocations.
– Embedding serialized data in dropped images.

We address these challenges in the remainder of this section.

Creating Toolglasses. We have implemented toolglasses in Mac OS X using the Cocoa Human Interface Device APIs. When a toolglass is first created, the list of available input devices is polled. By default, when more than one mousing device is present, the first one is bound to the toolglass and the remaining mice control the standard system pointer. Thus, if only one mouse is present, toolglass interaction is disabled, falling back on traditional interactions. On laptops, the builtin trackpad is generally the first mouse found and is bound to the toolglass.

The toolglass itself is an undecorated, focus-less window with a mostly transparent background. It is then filled with buckets, which are simply standard widgets (`NSView`), with all input redirected into a state machine using a Scotty event funnel [10]. In order to pass events to the underlying window, the toolglass clones the currently handled event and injects it into the system event queue (using a Cocoa event tap). That event can then be either consumed or propagated to the standard event stream. It is essential that the state machine for the toolglass preserve standard event symmetry invariants, such as press/release pairing and that drag events always occur between a press and a release. If these invariants are not respected, the program may enter into an undefined state.

Widget Design Patterns: (Controllers, Target-Action). Cocoa applications are typically developed using a model-view-controller (MVC) architecture [24], where event handlers are typically implemented in a controller object associated with the interface. Widgets, in turn, use a target-action pattern to designate their controller (the target) and the message[2] to send to the target (action). Other toolkits typically use a similar mechanism to handle callbacks, signals, or event handlers: when a widget is triggered, the system executes some bit of code with an event describing what happened and what was triggered. In Cocoa, when a widget is triggered, such as by clicking a button or pressing return in a text field, the widget sends its action message to its target with itself as the sole parameter: the sender.

Grab: Extracting a Widget's Action. Grabbing a widget's "callback" thus entails simply storing a reference to its target and action, and storing a reference to the widget itself. We also create a copy of its backing image in order to store its on-screen representation in the toolglass bucket. Invoking a previously grabbed command whenever it is dropped through the toolglass is then as simple as invoking the *action* on the *target* with the source widget as the *sender* (*e.g.*, `target.invoke(action, from: sender)`). This action will then typically operate on the user's current selection in the application.

[2] Objective-C uses message-passing to invoke methods.

Special Cases: State, Subclasses, Side-Effects. Some controls may be stateful. For example, a compound control, such as a list or a segmented toolbar control, may perform some form of "backchannel" communication beyond the standard target-action invocation model. An action method might, for example, query its sender to identify which list item or segment was clicked.

For such stateful controls, it is necessary to capture that state. We create a proxied copy of the control to be used as the sender in place of the original widget. This proxy is a Scotty object proxy [10]. An object proxy is a *meta*class[3] whose instances are subclasses of the class of the object to be proxied. An instance of this metaclass stores a copy of the source object to be proxied. Methods invoked on it can then be forwarded to the underlying object, creating a sort of man-in-the-middle stand-in for the original widget. We have implemented proxy classes for the standard outline/list view, popup button, and segmented control classes.[4]

When invoking a control, the toolglass uses this proxy as the sender instead of the original control. Thus, if the action method queries the control for one of these overridden methods, the captured state is used instead of the current state.

Drop: Invoking a Stored Action. When the user performs a drop, she clicks through the toolglass, possibly dragging, before eventually releasing. Toolglass buckets may be configured to trigger their action either before passing the initial click to the window below, as in the case of a command mode (*e.g.* the redeye tool), or after the terminating mouse release, as in the case of an operation on a selection (*e.g.* the styles chooser).

As shown above, actually triggering the callback associated with the source widget is straight-forward and simply involves sending the action message to the target with either the source widget or its proxy as the sender parameter.

Serializing the Grab 'n' Drop Toolglass. Initially identifying a widget, its target, and its action during a grab operation is easy: they come from the clicked widget. But in order to serialize toolglasses across invocations of the application, we need a way to find these same objects in a new application instance, without a user's click to identify the widget. Furthermore, when loading a previously serialized toolglass, the program may be in a different state. The interface may be in a different configuration and the relevant objects might not have been created.

During serialization, we need to construct a path from a known, fixed reference point to the widget. To construct this path, we use three starting points:

[3] A metaclass is a class of classes. Whereas an instance of a class is an object, an instance of a metaclass is itself a class.

[4] The first two simply override the `clickedRow` and `selectedItem` methods, respectively, to return whatever value was active when the control was grabbed. For segmented controls, there are two methods that need to be overridden: `selectedSegment` and `isSelectedForSegment:`, which are actually independent methods that handle multiple segments.

the current document, the current window, and the application's master controller. We then use the reflection APIs to conduct a breadth-first search using the attributes of each of these controllers to find a shortest path to the target objects. If none of the controller's instance variables matches the target widget, we expand the search to any controllers or delegate objects that it may reference. Unfortunately, controllers and delegates are merely design patterns; there is no formally-expressed relationship in the code itself. By convention, however, in Cocoa, such attributes typically have names ending in delegate or controller, as in `windowDelegate` and `playbackController`. Other conventions apply in other toolkits. We consider any such non-nil variables to be valid search candidates.

Finally, we fall back to the view hierarchy itself. We avoid using the view hierarchy because it is more likely to change between application launches. Nonetheless, it is sometimes the only available path to a particular controller. As such, any time we encounter an `NSView` instance, we add that to the search.

If no valid path is found at the end of this breadth-first search, it will not be possible to serialize that particular control for future use. In such a case, the bucket can still be used during the current session, but it will not be possible to save it and restore it later. If there are multiple paths, we use the first (shortest) path, preferring to avoid any paths involving the view hierarchy if possible.

In order to avoid unforeseen side effects, we look only at instance variable when searching for serialization paths. Comparing the values of two variables is safe. We ignore methods, since they may have unanticipated effects. For example, in some applications, calling the accessor for the current print job could potentially trigger a print job if one does not exist.

When loading in a new application instance, however, those attributes might not yet have been initialized. Thus, when deserializing, we prefer to use those very same accessors that we avoided during serialization.

In Java and other languages, accessors typically start with `get`. In Cocoa, methods and attributes are in different namespaces. We therefor look for a method with the same name, or with any leading underscores or a single `m` removed and re-camel-cased. This strategy relies on programming conventions and is thus inherently fragile.

Embedding Serialized Data in Dropped Images. Once the functionality for serializing a bucket between applications exists, embedding that serialized data in an image is easy. Furthermore, during a grab operation, we store a screenshot of the target widget.[5]

6 Discussion

6.1 Leveraging Design Patterns: Controllers, Target-Action

In the general case, giving the user complete control over all aspects of the user interface and functionality is almost certainly infeasible. Software applications

[5] The current implementation uses Cocoa's `CGImageProperties` APIs to read and write the serialized bucket into the screenshot's metadata.

are frequently complex, and user interfaces particularly so. Nonetheless, certain common design patterns in the development of user interfaces are particularly helpful at enabling the runtime modification of interaction and functionality. We rely on three design patterns in particular:

– The Model-View-Controller (MVC) programming model,
– Callback-bindings for widgets, and
– Dynamic dispatch in object-oriented programming languages.

The first of these, *MVC* [24], reflects the separation between interface, application functionality, and the link between the two. Views, or widgets, are typically programmed using standard classes in a toolkit library, although they are frequently customized through inheritance. The model is effectively opaque and may even be implemented in another programming language, but it is linked to the interface through the controller, which maps widget events to operations on the model. Different programs follow more- or less-clean implementations of this separation, and frequently the *PAC* model [5] under the guise of MVC. Regardless of the particular implementation, analyzing the controller provides a foothold into the underlying functionality of the program, and the links into the operational interface that controls it.

Callback-bindings are the linking mechanism between a widget and the controller. In Cocoa, these are implemented using the *target–action* model where a widget event, such as a button press, triggers an *action* on its associated *target*. The target is typically the associated controller, and the action is the associated method. The grab ‘n’ drop toolglass relies on being able to thus map widgets to their associated controllers and event handlers. Other user interface toolkits use different design patterns to express callbacks, but the overall result is similar: linking a widget to an event handler on some object. In Java Swing, this link is performed using a *listener* pattern; in Qt (C++), this binding is performed using *signals* and *slots*.

The last of these patterns, *dynamic dispatch*, is the process by which the programming language chooses, at runtime, what code to execute when a program invokes a method on an object. In most object-oriented programming languages, such as Objective-C, Java[6], C++[7], JavaScript, and Python, a dynamic component performs a mapping between an object and the particular method to execute. Our proof-of-concept implementation works by analyzing these mappings for a particular class to create an object proxy capable of standing-in for an existing object but with an overloaded or overridden implementation. This pattern is not strictly necessary to implement user-configurable toolglasses. If the target-action model were extended to provide for target-action-state, then applications could express their functionality entirely with such components. There would no longer be a need for proxy objects to, *e.g.*, hard-wire the selected item in a list or segmented button.

[6] for non-`final` methods.
[7] for all `virtual` methods.

6.2 The Interface as a Lens into the System

The human-software interface, of its nature, is designed to be understood by the human user of the software. As such, the interface exposes the core functionality of the system in a human-understandable fashion. While the design of the grab 'n' drop toolglass enables the user to transform the interaction around which the system was designed, it does not require a deep understanding of that system. Instead, it draws upon the user's understanding of what different controls do, of the user's own situated context, to re-assemble the underlying functionality into new controls. The only system knowledge necessary to implement the toolglass is that of the general design patterns described in the previous section.

More generally, other tools beyond the grab 'n' drop toolglass may be able to take similar advantage of the human user's understanding of the interface to make extensive modifications to the underlying functionality possible. The grab 'n' drop toolglass does so without programming, but tools such as Yeh *et al.*'s Sikuli [28] offer a glimpse into ways that end-users might be able to reprogram existing software applications, creating a sort of situated macros or software extensions to existing application interfaces. Sikuli relies exclusively on the surface representation of the application, but a hybrid approach that offers a similar degree of high-level interaction with existing components combined with direct access to their underlying functionality offers an exciting potential.

On the more advanced end of the spectrum, programmable tools in the spirit of Maclean *et al.*'s Buttons [21,25] could help make it possible for richer expressivity with underlying software functionality. With a sufficiently scaffolded environment, many kinds of programming tasks may become accessible even to end-user programmers [22].

6.3 End-User Programming

The grab 'n' drop toolglass aims to provide a programming-free means of letting a user adapt an existing user interface to her own needs. Tools such as Sikuli [28], Prefab [8], and Scotty [10], among many others, provide tools for programmers to alter software interfaces or functionality. Between these two ends of the spectrum lie more advanced end-user programming techniques, aiming to increase raise the ceiling of expressivity while maintaining a low threshold to entry.

For example, by using a model of intentionality based on some combination of the user's context and the widgets he or she grabs into the toolglass, the toolglass could potentially provide more seamless interaction without explicit programming. For richer control, EUD environments such as Sikuli could enable the user to go beyond simply extracting existing widgets to including new interactions and functionalities. Under the current proof-of-concept implementation, the user explicitly configures each bucket to control only the order of operations for dropped actions.

6.4 Design Opportunities

Under normal usage, an application’s interface provides its own set of widgets—which may potentially be instruments—through which the user interacts with associated domain objects. For most typical users, this exposed interface should be sufficient most of the time (assuming any reasonably well-designed application). For those users or those situations where that interface is not sufficient, however, the user may wish to act not on the domain objects but rather with the application itself. We have created the grab ‘n’ drop toolglass as a first step in this direction: the creation of a class of interactive tools to allow an end user to operate on an application’s underlying interface and functionality, similarly to how Façades let a user modify an application’s interface [26].

This type of interaction is common in physical interactions with real-world tools, where a master craftsman might typically use standard off-the-shelf tools to practice her trade. But when confronted with a particular need, she might create an adapter, or modify the tool in some way to better satisfy her special case. Should that adapter or modification be particularly useful, she might share it with others, possibly even becoming a new member of a standard toolbox, as we have seen with a wide variety of now standard tools: needle-nose pliers, 90-degree screwdrivers, *etc.*

This type of interaction offers an exciting design opportunity to create a broad class of tools. One need not necessarily look as far as the real world to identify these kinds of practices. Unix users have a long and well-documented history of customizing their software with Unix dotfiles [19], whether for functional reasons, to work around compatibility issues, or even for self-expression. Furthermore, sharing cultures [12,17] arising around such customizations have even led to entire communities, with customizations being produced and shared by expert programmers and tinkerers alike [21].

Furthermore, when confronted with a relevant problem within one’s domain, with sufficient motivation, even regular end-users may program extensive customizations for their own particular needs. For example, spreadsheet software has led to vast collections of complex logic programmed by secretaries and other non-computing professionals [22], while artists have formed communities around the development and exchange of Photoshop plugins and customizations [9].

6.5 Generalizability Across Applications

In order to understand the generalizability of the grab ‘n’ drop toolglass, we conducted a survey of different applications in addition to Aperture, Pages, and Mail, used in the scenarios: Keynote, Pixelmator, OmniGraffle, and Preview.

While the grab ‘n’ drop toolglass is able to bring toolglass interaction to applications that were not designed with such interaction in mind, not all interactions are compatible. Moreover, current designs of applications that would be amenable to toolglasses, *e.g.*, drawing applications, may use an unsuitable interaction vocabulary.

The grab ‘n’ drop toolglass works well with command-then-action or action-then-command interactions, such as toolbar buttons that operate on a selection. However, many buttons may not interact directly with a selection. For example, drawing in Keynote would be a perfect application for a traditional toolglass. However, its interaction model is such that, when the user creates a shape, it is dropped on the canvas at an arbitrary position. The user then resizes and re-styles that object. As such, it is possible to click through the toolglass to create a shape, but that shape is placed independent of where the user has clicked. As such, the user can compose a personal toolglass, but it’s utility degrades to that of a toolbar.

Similarly, OmniGraffle allows the user to click and drag out rectangles, but not other forms. The user must draw a rectangle, then change its form with a separate control. Moreover, it provides a list of forms that have been used in that document. As such, the list of pertinent forms depends on the current document and is thus inconsistent throughout the application. In this case, the user may expect to draw a circle, but instead find a cloud or a rounded rectangle because the captured state varies between documents.

On the other hand, applications with a more traditional modal palette interaction style, such as Pixelmator for drawing or Preview’s annotations toolbar, provide compatible interface elements. Creating, for example, an annotation toolglass with differently-colored highlighters or an underline thus behaves as expected.

Finally, our proof-of-concept is implemented as a hack on the system using Scotty [10]. As each new version of Mac OS introduces stricter sandboxing restrictions, fewer and fewer applications remain compatible. While this will prevent deployment of the prototype as-is, our contribution is an interaction technique for user-composable toolglasses, not a hack on Cocoa. For example, if Cocoa provided its own grab ‘n’ drop toolglass, there would be no need to break into the sandbox and into an application’s runtime.

6.6 Limitations and Fragility

We have successfully used grab ‘n’ drop toolglasses in a variety of Mac OS X Cocoa-based applications, including Pages (word processor), Keynote (presentation tool), Mail, Aperture (photo manager), and others. Additionally, we have used the toolglass on a variety of widgets, both standard and custom, including standard buttons (*e.g.* OK), toggle buttons (*e.g.* on/off switch), segmented buttons (*e.g.* unified bold/italic/underline button), pull-down menus (*e.g.* font button), and lists.

Nonetheless, any time one modifies the behavior of an existing program, especially without knowledge of its underlying design and implementation, there is a non-insignificant risk of breaking any hidden assumptions and thus introducing instability. Such instability may range from mild, such as a user expecting an action to make something bold but seeing it turn italic instead, to severe, such as corrupting or losing data or crashing.

In our use of the grab 'n' drop toolglass, we have not observed data corruption. The worst we have seen has been unexpected program termination (*i.e.* a crash) induced by a bug (since corrected) in the implementation of the toolglass prototype that injected an extraneous mouse press event or suppressed a mouse release event, breaking the system's underlying assumptions about the symmetry of press/release events.

The most common sort of non-bug-related unexpected behavior that we have observed in our testing relates to uncaptured state leading to hidden behavior not exposed by the standard callback model. As we have seen earlier, in the standard target-action design pattern, a widget invokes its target's action method with itself as the sole parameter, the sender, whenever a widget event is triggered (*e.g.* a button is clicked). For a button, this is the end of the widget's involvement, but for other widgets, such as a list or a segmented control, the action method *may* query the sender for its state, such as the item selected. Or it may even interact with any other arbitrary part of the system to determine its behavior. We are not aware of a robust, automatic solution to capture and model such hidden interactions. If the behavior of a triggered action depends on such hidden interactions, then the toolglass might end up triggering a different, unexpected behavior.

We have implemented proxies for the following standard widgets:

- `NSOutlineView` to capture the clicked row
- `NSPopUpButton` to capture selected item
- `NSSegmentedControl` to capture the selected or clicked segment.

The grab 'n' drop toolglass uses these proxies any time one of these widgets or any of its subclasses is grabbed. Although our current implementation has worked in our usage, there is a risk that a subclass may only re-use other aspects of the parent class' behavior and may re-implement or bypass the captured behavior of the proxy. In these cases, the resulting behavior is difficult to predict.

We have also created a `SNDTracerProxy` that logs all method calls to the console before forwarding them to the proxied object. This proxy is useful for a programmer to probe any hidden interactions that may involve the proxied object but is beyond the scope of normal user interaction with the toolglass.

It may be possible to resolve much of this fragility by extending the target-action design pattern to more fully express some of these relationships. Such extensions, however, would require that programmers change the way that they develop software. Furthermore, if these patterns do not fully integrate with the tools to implement the software functionality, then they could add a similar burden as AppleScript [1] or other extension interfaces, which provide a second, parallel API for developers to maintain.

Even with strong design patterns, not all software is well-designed. Certain applications do not adequately follow recommended design patterns or even misuse them, resulting in applications that may be difficult to maintain even with access to their source code. Augmenting their existing behavior or interaction may further exacerbate such fragility.

7 Conclusions and Future Work

The grab ‘n’ drop toolglass provides a relatively simple interaction technique through which end-users can modify the underlying interaction of existing applications without access to their source code. In many cases, it lets user create new, customized toolglasses suited to their own particular needs without programming and without special support by the underlying application. We have demonstrated a proof-of-concept implementation for Mac OS X Cocoa applications, but variations of this technique should be compatible with other graphical environments.

We view the grab ‘n’ drop toolglass as part of a larger class of user-programmable interactions, wherein the underlying software functionality and interaction is malleable. We plan to continue our work in this area, investigating a combination of novel programming models and interactions techniques to increase the flexibility and control over the software without requiring extensive programming on the part of the user. Of particular need are richer design patterns to better express the currently-hidden assumptions behind the interactions between the user interface and the application functionality.

Additionally, there is currently no effective way for an end user to adequately gauge the fragility and risk associated with performing certain operations. It may not be fully necessary to fully insulate the user from all risk so long as she may be able to make her own informed decision as to whether to assume it. In future work, we plan to explore such trade-offs.

References

1. Apple Computer Inc.: AppleScript Language Guide. Addison-Wesley Longman Publishing Co., Inc., Boston (1994)
2. Beaudouin-Lafon, M.: Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In: CHI 2000: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 446–453. ACM, New York (2000)
3. Beaudouin-Lafon, M., Mackay, W.E.: Reification, polymorphism and reuse: three principles for designing visual interfaces. In: AVI 2000: Proceedings of the Working Conference on Advanced Visual Interfaces, pp. 102–109. ACM Press (2000)
4. Bier, E.A., Stone, M.C., Pier, K., Buxton, W., DeRose, T.D.: Toolglass and magic lenses: the see-through interface. In: SIGGRAPH 1993: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, pp. 73–80. ACM, New York (1993)
5. Coutaz, J.: PAC: an object oriented model for implementing user interfaces. SIGCHI Bull. **19**(2), 37–41 (1987). http://doi.acm.org/10.1145/36111.1045592
6. Cypher, A., Dontcheva, M., Lau, T., Nichols, J.: No Code Required: Giving Users Tools to Transform the Web. Morgan Kaufmann Publishers Inc., San Francisco (2010)
7. Cypher, A., Halbert, D.C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B.A., Turransky, A. (eds.): Watch What I Do: Programming by Demonstration. MIT Press, Cambridge (1993)

8. Dixon, M., Fogarty, J.: Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In: CHI 2010: Proceedings of the 28th International Conference on Human Factors in Computing Systems, pp. 1525–1534. ACM, New York (2010)
9. Dorn, B., Tew, A.E., Guzdial, M.: Introductory computing construct use in an end-user programming community. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, VLHCC 2007, pp. 27–32 (2007). http://dx.doi.org/10.1109/VLHCC.2007.33
10. Eagan, J.R., Beaudouin-Lafon, M., Mackay, W.E.: Cracking the cocoa nut: user interface programming at runtime. In: Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, UIST 2011, pp. 225–234. ACM, New York (2011). http://doi.acm.org/10.1145/2047196.2047226
11. Fishkin, K., Stone, M.C.: Enhanced dynamic queries via movable filters. In: CHI 1995: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 415–420. ACM Press/Addison-Wesley Publishing Co., New York (1995)
12. Gantt, M., Nardi, B.A.: Gardeners and gurus: patterns of cooperation among cad users. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 107–117. ACM, New York (1992)
13. Hudson, S.E., Rodenstein, R., Smith, I.: Debugging lenses: a new class of transparent tools for user interface debugging. In: UIST 1997: Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology, pp. 179–187. ACM, New York (1997)
14. Hutchings, D.R., Stasko, J.: Quantifying the performance effect of window snipping in multiple-monitor environments. In: Baranauskas, C., Palanque, P., Abascal, J., Barbosa, S.D.J. (eds.) INTERACT 2007. LNCS, vol. 4663, pp. 461–474. Springer, Heidelberg (2007). doi:10.1007/978-3-540-74800-7_42
15. Kato, J., Sakamoto, D., Igarashi, T.: Picode: inline photos representing posture data in source code. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2013, pp. 3097–3100. ACM, New York (2013). http://doi.acm.org/10.1145/2470654.2466422
16. Klokmose, C.N., Zander, P.-O.: Rethinking laboratory notebooks. In: Lewkowicz, M., Hassanaly, P., Wulf, V., Rohde, M. (eds.) Proceedings of COOP 2010, pp. 119–139. Springer, London (2010). doi:10.1007/978-1-84996-211-7_8
17. Mackay, W.E.: Patterns of sharing customizable software. In: Proceedings of the 1990 ACM Conference on Computer-Supported Cooperative Work, pp. 209–221. ACM Press, New York (1990)
18. Mackay, W.E.: Users and Customizable Software: A Co-Adaptive Phenomenon. Ph.D. thesis, Massechusetts Institute of Technology (1990)
19. Mackay, W.E.: Triggers and barriers to customizing software. In: CHI 1991: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 153–160. ACM Press (1991)
20. Mackay, W.E.: Which interaction technique works when?: floating palettes, marking menus and toolglasses support different task strategies. In: AVI 2002: Proceedings of the Working Conference on Advanced Visual Interfaces, pp. 203–208. ACM, New York (2002)
21. MacLean, A., Carter, K., Lövstrand, L., Moran, T.: User-tailorable systems: pressing the issues with buttons. In: CHI 1990: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 175–182. ACM Press, New York (1990)
22. Nardi, B.A.: A Small Matter of Programming: Perspectives on End User Computing. MIT Press, Cambridge (1993)

23. Ponsard, A., McGrenere, J.: Anchored customization: anchoring settings to the application interface to afford customization. In: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, CHI 2016. pp. 4154–4165. ACM, New York (2016). http://doi.acm.org/10.1145/2858036.2858129
24. Reenskaug, T.: Models—views—controllers. Technical report, Xerox PARC, December 1979
25. Robertson, G.G., Henderson Jr., D.A., Card, S.K.: Buttons as first class objects on an X desktop. In: Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology, pp. 35–44. ACM Press, New York (1991)
26. Stuerzlinger, W., Chapuis, O., Phillips, D., Roussel, N.: User interface façades: towards fully adaptable user interfaces. In: UIST 2006: Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology, pp. 309–318. ACM, New York (2006)
27. Tan, D.S., Meyers, B., Czerwinski, M.: WinCuts: manipulating arbitrary window regions for more effective use of screen space. In: CHI 2004 Extended Abstracts on Human Factors in Computing Systems, pp. 1525–1528. ACM, New York (2004)
28. Yeh, T., Chang, T.H., Miller, R.C.: Sikuli: using GUI screenshots for search and automation. In: UIST 2009: Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology, pp. 183–192. ACM, New York (2009)