Patina: Dynamic Heatmaps for Visualizing Application Usage

Justin Matejka, Tovi Grossman, and George Fitzmaurice
Autodesk Research, Toronto, Ontario, Canada
firstname.lastname@Autodesk.com

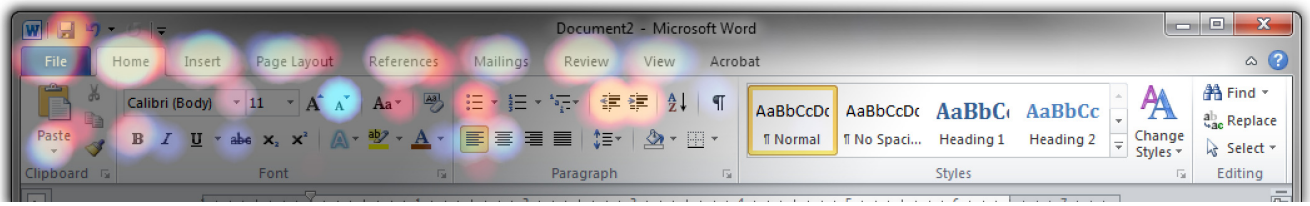


Figure 1. A side-by-side view of the Patina heatmap overlay showing the usage patterns of both the active user and the user community on the left, and the standard underlying Microsoft Word interface on the right.

ABSTRACT

We present *Patina*, an application independent system for collecting and visualizing software application usage data. Patina requires no instrumentation of the target application, all data is collected through standard window metrics and accessibility APIs. The primary visualization is a dynamic heatmap overlay which adapts to match the content, location, and shape of the user interface controls visible in the active application. We discuss a set of design guidelines for the Patina system, describe our implementation of the system, and report on an initial evaluation based on a short-term deployment of the system.

Author Keywords: Visualization; Social Learning

ACM Classification: H.5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

INTRODUCTION

In today's software applications, users can be faced with thousands of menus, dialogs, and interactive widgets, making the usage and navigation through those interfaces overwhelming. These applications typically look the same regardless of their past usage. A user will be faced with the exact same user interface, regardless of how many times it has been used. On the contrary, physical objects give people a rich set of cues related to their usage history; We can recognize that a car is brand new by its smell, that a book has been well read by the deteriorating visual appearance of its cover, or that a baseball glove has a long history of use from its feel, and the ease at which it closes.

Pirolli and Card's information foraging theory [22] introduced the concept of *information scent*, defined as "the (imperfect) perception of the value, cost, or access path of information sources obtained from proximal cues." Research has shown that the existence of information scent can aid in navigation and decision making tasks [27]. Thus, improving the information scent cues in software application interfaces could help alleviate the challenges imposed by their overwhelmingly large feature sets.

One way to provide information scent in a user interface is to visualize cues related to the history of its usage [27]. In their Scented Widgets paper, Willett *et al.* argue that such social navigation cues can "direct our attention to hot spots of interest or to under-explored regions." For example, a user opening up an advanced preference dialog may be able to quickly identify settings that users rarely disabled, or parameters that are commonly adjusted.

While usage metrics for software applications can often be collected [17,19], doing so typically requires instrumentation of the host application. Similarly, supporting scented widgets [27], or adapting an application to a user's past behaviors [12], requires modification of the application.

In this paper, we present *Patina*, a new system that collects and visualizes software application usage data. Our system adds two core contributions to the existing literature.

First, whereas scented widgets were designed to enhance individual or groups of widgets, Patina provides visual cues across an entire application interface using a dynamic graphical overlay. A colored heatmap indicates commonly and rarely used features in any view of the interface, and adapts to the current interface layout. Second, Patina is implemented in an application-independent manner requiring no instrumentation of the host application, both for the collection *and* presentation of the usage metrics. This is made possible using a combination of system window metrics and accessibility information, available in many of today's Windows applications.

In the following sections, we provide an overview of the related research, discuss the design goals of our system, and present the implementation details of Patina. We also report on an initial evaluation based on a short-term deployment of the system, used by 8 users for 1 week.

Our results and experiences indicate three primary scenarios where the Patina system could be useful: familiarizing new

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CHI 2013, April 27–May 2, 2013, Paris, France.
Copyright © 2013 ACM 978-1-4503-1899-0/13/04...$15.00.

users with an application; exposing functionality relevant to a specific document; and supporting continuous reflection on personal and community usage patterns.

RELATED WORK

Collecting and Visualizing Usage Metrics

Many commercial software products have customer usage reporting facilities which log and report usage metrics. The ingimp project [26] instrumented the open-source image editing program "the GIMP" to collect real-time usage and demographic information. As with commercial applications however, ingimp required modifying the original source of the application to collect usage data. The Patina system is designed to require no modifications to the original application. The AppMonitor tool [1] from Alexander *et al.* is a client side logging tool that records user interactions in unmodified Windows applications. However, a special DLL must be loaded into a shared memory space with the target application. In contrast, the Patina system collects and monitors data only through an external process. Hurst *et al.*'s Dirty Desktops [16] identified likely interface targets through the collection of user click points but relied on a fixed size and arrangement of UI elements, while the Patina system is designed to work with resizable UIs.

Semi-transparent heap-maps overlaid over the original content are frequently used for viewing eye tracking data [28]. Heatmap overlays have also become a popular way for website administrators to view where user's click on their website [29]. These displays can be useful, however they are not robust to changes in the layout of the underlying webpage. The Mozilla Labs team instrumented the Firefox browser to collect data on which interfaced regions were clicked. They then plotted the results over a static image of the browser [30], whereas the Patina system displays a live-updating heatmap over a running application.

Information Scent

Many research projects have looked to improve the information scent [22] of an interface to assist in the user's navigation through or usage of a software application. In 1992, Hill and Hollan [14] introduced the idea of *computational wear* by marking up the scroll bar of a text editor with indications of which parts of the document have been frequently read and/or edited. This idea was also explored more recently by Alexander *et al.* [2]. Patina employs the idea of *computational wear* by essentially marking up areas of the UI based on their usage.

Scented Widgets [27] offers visual encodings built into individual UI widgets to show community gathered usage data. In contrast, Patina visualizes social navigation cues over the entire application, using a dynamically generated heatmap, and introduces an application independent implementation of Scented Widget visualizations.

The Phosphor [4] and Mnemonic Rendering [5] systems use visual feedback to attract the user's attention to settings or parts of the screen which have been modified. This might allow a user to see which parts of the interface are more useful or important, and the Patina system can serve a similar purpose.

Adaptive UIs

Researchers have explored several ways to address the issue of a user being overwhelmed by the multitude of options and tools available in a complicated software application. One approach has been a multi-layered, or "training wheels" interfaces [3, 6] which only expose new users to a subset of the available functionality, and gradually expose more functionality as the user becomes more experienced. These techniques can reduce the number of mistakes made by a user, but are not applicable for experienced users, or in situations where the user really does need access to the full functionality of the system.

To provide the benefits of multi-layered interfaces while still providing access to the full range of functionality, adaptive menus [12, 13] have been explored which automatically rearrange the items in a menu placing the most frequently used items at the top. These techniques suffer from rearranging the items in the interface, requiring the user to "re-find" elements which have been displaced. Findlater *et al.*'s Ephemeral Adaptation [13] addressed this shortcoming of adaptive menus by maintaining a fixed menu item arrangement where the predicted items appear immediately while the remaining items fade in after a short (500ms) delay. The Patina system uses an automatic transient display similar to Ephemeral Adaptation.

UI Recognition

For a system to augment the interface of an existing application without access or modifications to the original source code, it must be able to recognize the location and properties of the application's UI elements. Prefab [10, 11] and Sikuli [8] use a vision based approach to locate user interface elements based on their appearance. Systems by Hurst *et al.* [15], the PAX framework [7], and Façades [24] combine image techniques with accessibility data collected from the publicly exposed accessibility APIs. Our system exclusively uses externally available window and accessibility data but could be made to take advantage of additional recognition techniques.

While aspects of our design have been inspired by previous work, our system is unique and flexible. None of the previous systems offer an application-independent means of visualizing software application usage data via dynamic graphical overlays which adapt to match the content, location and shape of the user interface controls.

DESIGN GUIDELINES

Our design of Patina was grounded by a study of related research and theory on information visualization and information scent. Below we describe the guidelines that we followed in our design process.

Uniform: Encoding the same data in different ways across widgets can complicate visual comparison [27]. As such, the visual encoding should be consistent across the entire user interface.

Distinguishable: Information scent encoding should not conflict with the conventions of the underlying content [27]. Our visual encoding should respect and be *easily distinguishable from the underlying interface conventions.*

Intuitive: For a visualization to be effective, the user must be able to understand it. Users should be able to, without training, see which areas of the interface are heavily used, as well as which areas are infrequently used. It is therefore important to use a visual encoding that will be intuitive for users to comprehend.

Proximal: To aid in navigation of an information space, the information scent should be provided in the form of a proximal cue [22, 27]. Our visualization should be presented in the same visual space as the user interface elements so the mapping between UI element and usage is directly visible.

Non-Disruptive: The information scent should not adversely impact the user interface design or layout [27]. The data presentation should strike a balance between providing useful information while not being disruptive.

PATINA SYSTEM DESIGN

Our goal is to design a system that works without any modifications to the target application, and without any specialized knowledge about the internal workings of the application. Additionally, the system should be designed in a way that allows it to collect and display usage data from any application.

The Patina system is implemented in C# as a Microsoft Windows application. The main system is broken down into two main sections of Data Collection, and Presentation, with a Data Management block in between (Figure 2).

Figure 2. Organization of the Patina System.

Data Collection

Typical heatmaps collect and visualize static *x* and *y* click points [29] or eye-tracking coordinates [28]. However, to implement a visualization on a live, resizable, customizable user interface, our system needs to recognize which user interface controls the user has interacted with. Our implementation uses Windows-specific libraries for collecting the necessary data, although similar functionality does exist for other operating systems.

Window-Level Data

Top-level, or main application windows in the Windows operating system are accessed programmatically through a handle to the window, referred to as an hWnd. Through a collection of Win32 API calls to functions (including `GetWindowText`, `GetClassName`, and `GetWindowThreadProcessId` hosted in the `user32.dll` file), a selection of information about the window and associated process can be gathered (Table 1).

The *Window Title* represents the text which appears in the title bar of the active window. If we are looking at the main application window, the *Application Window Title* field will be the same as *Window Title*. However, if we are looking at a dialog box of some other secondary window, the *Application Window Title* will have the text in the title bar of the host application.

PROPERTY	EXAMPLE
Window Title	**"Modify Style"**
Application Window Title	**"Docu2.docx - Microsoft Word"**
Location (x, y)	`408, 457`
Size (width, height)	`532, 545`
Module Name	`WINWORD.EXE`
Class Name	`bosa_sdm_msword`

Table 1. Information collected for a top-level window, the "Modify Style" dialog in Microsoft Word 2010.

In Windows there is also the notion of *control* windows, which are not windows in the traditional UI sense, but rather are sub-elements within a parent window such as scrollbars, informational status areas, or the main canvas area. Outlines for all areas defined as nested hWnd's from a standard view of Microsoft Word and AutoCAD are shown in Figure 3. The *Class Name*, *Size*, and *Location* are collected and logged for each new window activation. Each time the user performs a click event, we check the hWnd hierarchy to see if there have been any structural changes, and if so, we log the differences.

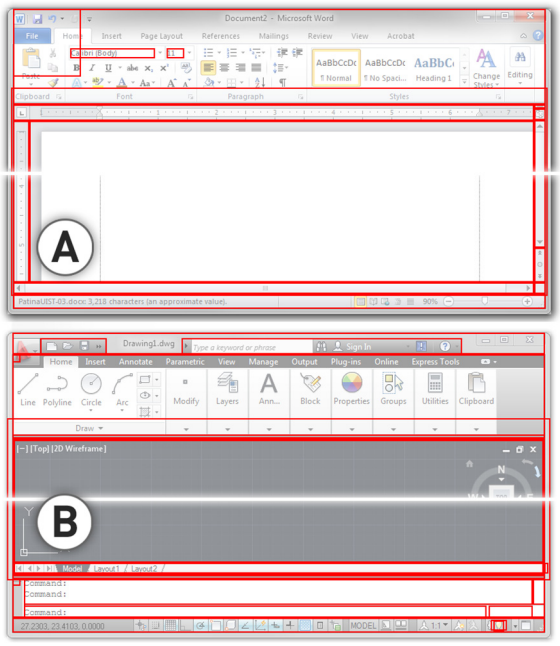


Figure 3. Rectangle information collected from HWND data structures from standard Microsoft Word (A) and AutoCAD (B) windows.

It is apparent by looking at Figure 3 that not all of the area information collected through the interrogation of the nested control windows corresponds visually with UI elements; for example, the square in the top left corner of the Microsoft Word window. Despite the somewhat dirty nature of the data

collected in this way, many elements are detected such as the scroll bars in Microsoft Word, and the tool pallets and command line area in AutoCAD.

Accessibility-Level Data

Accessibility APIs are interfaces included in many operating systems which provide programmatic access to user interface elements and are typically used by assistive technologies such as screen readers or GUI automation tools. It generally takes additional work from application developers to fully support the platform's accessibility API's, and as such, the completeness of accessibility coverage can vary greatly between applications. Hurst *et al*. [15] found that over a dataset of 1335 interface elements from 8 popular applications, the Microsoft Active Accessibility (MSAA) API was able to correctly recognize 74% of the UI targets. Our exploration has found that traditional UI elements such as buttons, scrollbars, combo boxes, menus, pull-down menus etc. are relatively well supported by the Accessibility API, but more specialized or unique controls are less reliably covered. Figure 4 shows the accessibility regions returned when querying the same two main windows from Figure 3. We can see that all of the standard controls have been recognized but several specialized controls have been missed; For example, the margin handles in the Word ruler bar, and the document tabs and in-canvas UI elements in AutoCAD.

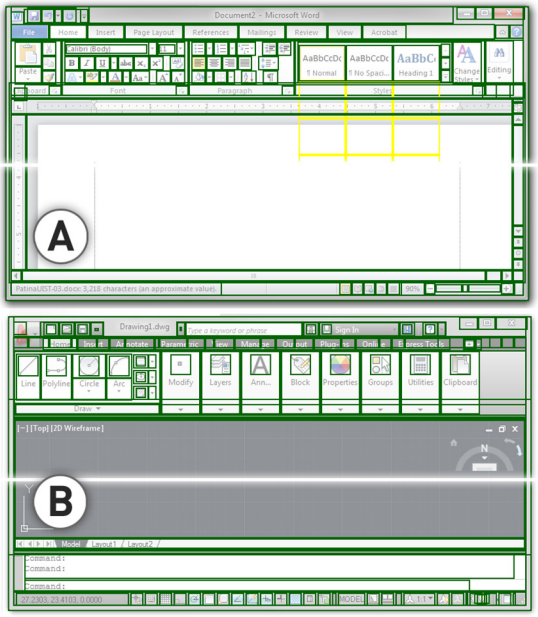


Figure 4. Rectangle data collected from the Accessibility APIs for Microsoft Word (A) and AutoCAD (B) windows. Yellow rectangles indicate regions which are reported as "offscreen".

When pull-down or pop-up menus are posted, new accessibility regions are generated for the individual items in the menu, and the Patina

Our system gathers accessibility data using the previously mentioned Microsoft Active Accessibility (MSAA) API through the Managed Windows API[1] wrapper. Each item exposes a different set of parameters through the API (as members of the `SystemAccessibleObject` class), but an example of the data available for a combo box is presented in Table 2.

The *Role* field contains what type of UI element we are accessing and the *State* field reports the current condition of the control, such as *offscreen* for items that are not currently visible, and *checked* for selected checkboxes. The current value of a UI element with a user-modifiable component such as a text field or combo box is reported in the *Value* field.

PROPERTY	EXAMPLE
Name	**Font:**
Role	**Combo Box**
State	**None**
Value	**Times New Roman**
Description	**Change the font face.**
Shortcut Text	`null`
Location (x, y)	`303, 83`
Size (width, height)	`98, 22`

Table 2. Information collected for a UI element with the Accessibility API, in this case, the font selection combo box in Microsoft Word 2010.

Querying a single accessibility object can be done without a noticeable delay, but requesting the entire accessibility object tree, which we require, is more intensive, so we perform this data collection in a background thread. Each time the user clicks, we query the accessibility API to get a listing of all available UI elements, and compare against the previously cached list of elements. If there are any additions or removals between the two lists, the new list is cached and saved to disk.

User Activity Data

Besides collecting identifying and structural information related to the window and UI components on the hWnd and accessibility levels, the Patina system is also notified of when a new foreground window is activated. Mouse click events are captured and the coordinates are saved relative to the coordinates of the active window.

Data Management and Sharing

Our system architecture utilizes Dropbox[2] as a mechanism to share data among users [19]. Collected window, accessibility, and user data is placed in a shared Dropbox folder and automatically synced with all other users. This technique simplifies the deployment and evaluation of the system, in comparison to commercial cloud based data management services that would be used for an actual implementation.

Presentation: Dynamic Heatmaps

The Patina system uses dynamic heatmaps as the primary mechanism for encoding usage information. Heatmaps were chosen after a consideration of our grounded design goals. First, heatmaps visualize usage data across the entire interface with a consistent visual encoding (*Uniform*). Heatmaps

[1] mwinapi.sourceforge.net

[2] www.dropbox.com

are also capable of being overlaid ontop of the user interface (*Proximal*). Furthermore, the organic nature of our heatmaps are clearly distinguishable from the interface itself (*Distinguishable*). Finally, heatmaps have become a common method for encoding web analytics [29] since it is intuitive for users to understand the meaning of the "hot zones" (*Intuitive*).

The presentation layer of the Patina system comprises two main functions: first, generating the usage pattern heatmap for the current view, and second, displaying the heatmap overlay on top of the active window.

Generating the Heatmap

The process of creating the heatmap can be broken down into four main steps as described below.

1 - Collecting Relevant Click Points

Since the Patina system works across applications, only a subset of previously recorded click points will have occurred in the current working application. The first step in filtering the entire set of mouse click points is to find those which occurred in the current application. We do that by considering the *Module Name* and *Class Name* fields from the window data which tells us the name of the executable and the type of window where the click occurred. We only consider clicks which were generated in windows matching the current *Module* and *Class Name*.

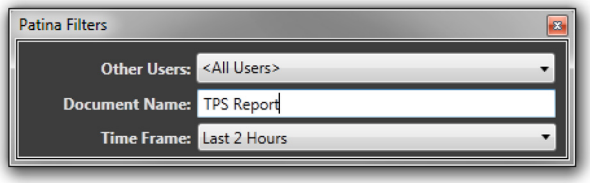


Figure 5. Filters dialog to specify parameters to narrow the data used to generate the heatmaps.

The *Filters Dialog* (Figure 5) provides an additional level of control over filtering the data. Users can choose to only consider clicks from a particular user, particular document (which is implemented as a filter on the *Application Window Title* field), or from a particular time frame.

2 - Mapping Click Points to Current Interface

At this point we have a collection of click points which may be relevant to the current view. The next step is to see which ones have a mapping to the application's current view.

In the simplest scenario, the original window where the recorded click occurred, and the current view, will be exactly the same; that is, they have the exact same dimensions and they have the exact same content. This could occur for a fixed-size window without any tabs or dynamic controls [16]. In this scenario we could simply use a static heat map, using the originally recorded click points. However, few such static user interfaces exists, so we look at our logged control window and accessibility data for a more generalized solution.

Through the collection of control window and accessibility regions we have a set of structural and organizational data about the state of the window at each past click event. We refer to these control windows and accessibility regions collectively as *control regions*. Some of these *control regions* are quite large and non-specific. For example, the "Home: property page" accessibility region represents the entire *Home* tab of the ribbon. Others are smaller and more precise, such as the "Bold: push button" which corresponds to the 23x22 pixel *Bold* button (Figure 6). The system preferentially uses the accessibility region data for UI element discrimination, and only uses the control windows when no accessibility information is available.

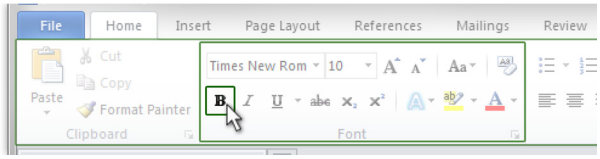


Figure 6. Overlapping rectangular accessibility regions for the "Bold: push button", with the larger areas being the "Font: toolbar" and "Home: property page".

Since these areas are nested and overlapping, the location of each click event could be within multiple *control regions*. To determine at the finest granularity which UI element the click occurred in we look for the smallest *control region* which contains the click coordinates, and associate that *control region* with the original click event. We then look for a corresponding *control region* in the currently active window. For accessibility regions we do this by looking for a region with matching *Name*, *Role*, and *Description* fields. If we find a match, we keep this click point and use it in the next step.

3 - Creating Intensity Map

At this point we have a collection of click points and associated *control regions* in their original context and we need to map them to the current display. Since the *control region* in the original capture and the matching *control region* in the current interface might have different locations and/or sizes, we consider the position of the click within the original *control region* relative to the width and height of the region, and map the same relative values onto the corresponding region in the current view (Figure 7).

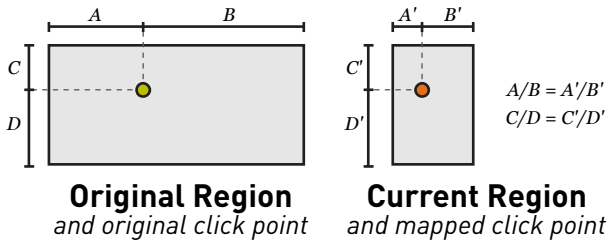


Figure 7. Click point mapping from original *control region* to current *control region*.

This relative positional mapping of click points allows the heatmap to maintain a correct view of usage patterns when UI elements have been moved around on the screen, as well as when the UI controls themselves change between different sizes such as resizing icons in a Ribbon toolbar. The mapping from the original click points to corresponding points on the current view is recalculated every time the current view changes and allows the heap map to update based

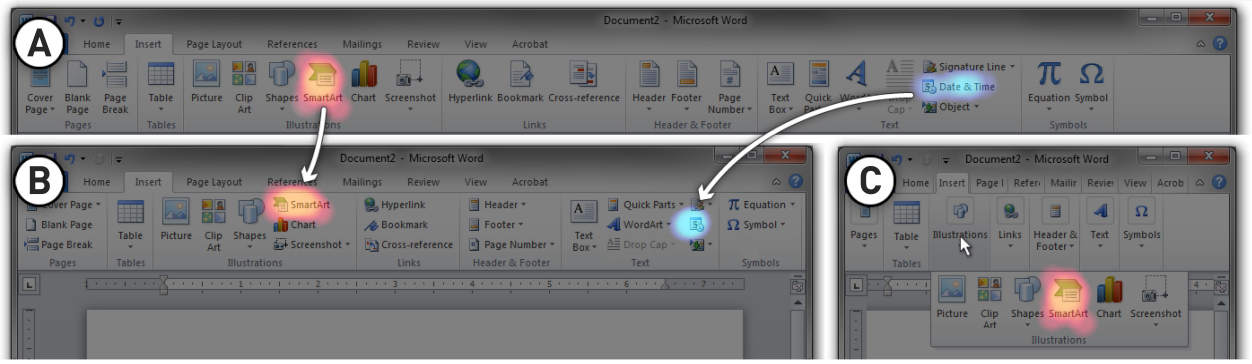


Figure 8. Demonstration of Patina overlay persisting on the correct UI elements after a window resizing, even when the target UI elements change size and position between (A) and (B). During ribbon resizing icons may become hidden (C), however when those elements are exposed through the fly-out menu, their Patina hotspots are restored.

on any changes to the interface layout. An example of this behavior can be seen in Figure 8.

Once all the click points have be mapped onto the current interface layout, the intensity map is created by drawing a semi-transparent circle at each click location which fades from its most opaque in the center to transparent at the edges (Figure 9). The size of the circle is configurable, but we used a radius of 20 pixels for the prototype. The base opacity for each click point is 40%, and as points overlap each other, sections of the intensity become darker, approaching solid black, indicating high activity. If there are many overlapping click points, the opacity of each point can be reduced to prevent the heatmap from becoming oversaturated. Initial testing found that many clicks points occur in the main canvas or working area of an application, and have a tendency to distract from the more useful data related to usage of the specific UI elements such as buttons. Since our main goal is to grack usage of interactive widgets, the opacity of individual click points are reduced for all accessibility regions with an area greater than 64x64 (4,096 pixels^2, which corresponds approximately to the largest size of buttons found in a Ribbon interface) down to 3% for all click points in a region larger than 90,000 pixels^2 (Figure 9A).

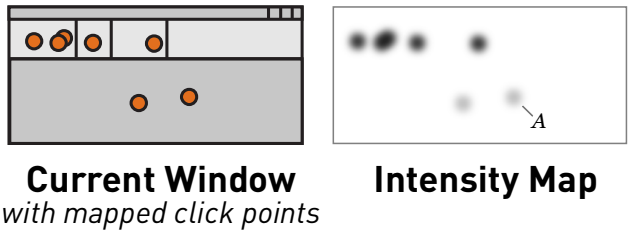


Figure 9. Intensity map creation.The intensity mapping process is done once for the active user's data and again for the rest of the community data.

4 - Coloring Heatmap.

Once the intensity maps are created they are converted into heatmaps. This conversion is done on a per pixel level mapping of the greyscale level of the intensity mask to an appropriate color (Figure 10). The heatmap for the active user is generated using the "You" band of colors on the left while the community heatmap is generated using the "Others" band of colors on the right.

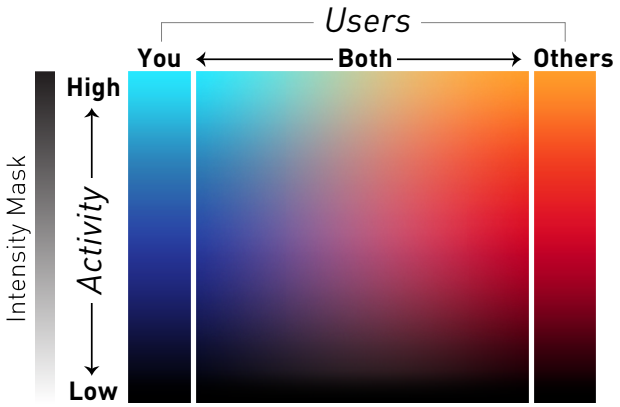
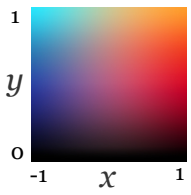


Figure 10. Color mapping used for the heatmaps ranging from low activity to high activity on the vertical axis, and the active user to the community on the horizontal axis.

This coloring creates the look of blue spotlights being used for highlighting the active user's usage data, and orange lights being used when displaying the community usage patterns (Figure 13). In addition to heatmaps showing only one of either the active user or community usage data, a third heatmap is created to create a combined overview. For this heatmap, the "Both" portion of the coloring chart is used, with the intensity level taken as the maximum of the two intensity masks at each pixel, and the color chosen as a blend between the two groups based on relative proportion of activity (Figure 11).



color(pixel **P**) = color_map(x, y)
where:
$x = intensity_{other}(\mathbf{P}) - intensity_{you}(\mathbf{P})$
$y = MAX(intensity_{other}(\mathbf{P}), intensity_{you}(\mathbf{P}))$
and:
$0 \le intensity(\mathbf{P}) \le 1$

Figure 11. Formula for determining the color of a given pixel in the combined heatmap.

We looked at several different schemes for coloring the heatmaps and found that this combination gave the best combination of visual appearance and ease of recognition when

looking at the *you* and *others* heatmaps individually, as well as when combined.

Displaying Resulting Heatmap

When a window is activated, a floating panel is positioned over the top left of the window giving information about the Patina system (Figure 12) including the state of data collection, the quality of information available for this application/window, and indicators to show which heatmaps are currently being displayed.

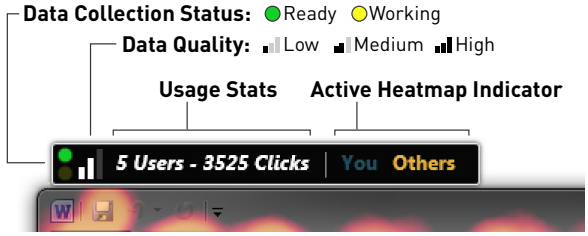


Figure 12. Patina information panel.

To maintain our *non-disruptive* deign goal, the primary method for viewing the heatmap is manually through hotkeys, F2 for the "You" heatmap and F3 for the "Others" heatmap. The "You" and "Others" indicators in the information panel can also be clicked. The individual heatmaps are displayed when only one of "You" or "Others" is selected, and the combined heatmap is displayed when both are chosen. The heatmaps smoothly fade in over a duration of 0.3 seconds and are displayed over the entire window at an opacity of 50%. An example of the three different heatmap views is shown in Figure 13.

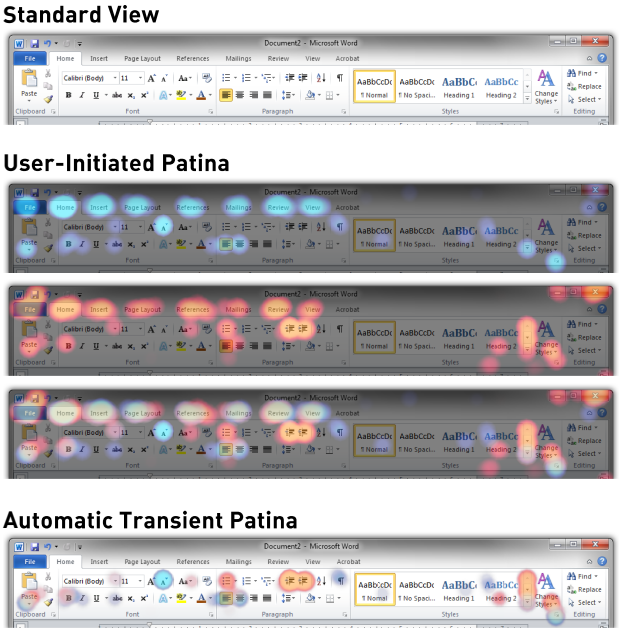


Figure 13. Examples of the different overlay modes.

An alternative visualization we considered was to render the usage patterns as rectangular overlays covering the extents of the UI widget (Figure 14). However, we prefer the organic look of the click-point representation, and believe it better satisfies our *Distinguishable* and *Intuitive* design goals.

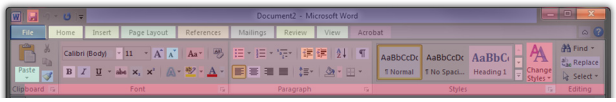


Figure 14. Combined usage overlay with rectangular regions matching over UI elements.

Automatic Transient View

Besides the user initiated display of the entire-window heatmap we have also created a view that is automatically and temporarily displayed when new UI components become visible (Figure 15). This mode gives the benefits of the Patina overlay without requiring the user to manually activate the visualization, and is similar in nature to Ephmeral Adaptation [13]. To minimize visual distraction, the heatmap is rendered with a transparent background, and only points associated with newly visible controls are included. For example, in the scenario shown in Figure 15, once the user clicked on the "Page Layout" tab, new UI elements appeared on the screen; namely, all of the controls under the "Page Layout" tab. Only these newly displayed controls are considered when gathering the points for this transient heatmap which smoothly fades in and out over a period of 5 seconds.

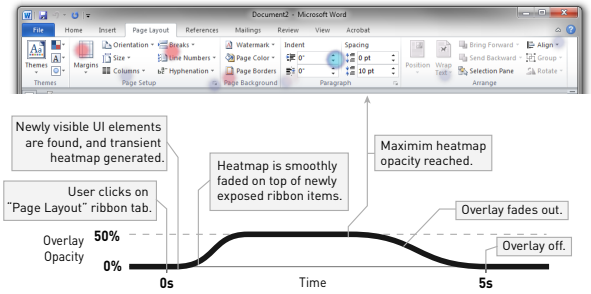


Figure 15. Visual example and time graph for the automatic transient Patina view.

This transient overlay view also works well to see which options are frequently modified when scrolling through large preference dialogs with many items.

Additional Applications of the Patina System

In addition to the previously described dynamic heatmaps, we now showcase how Patina can be used for application-independent implementations of three previously published research systems: *Scented Widgets*, *Usher*, and *CommunityCommands*.

Scented Widgets

Willett, Heer, and Agrawala's *Scented Widgets* [27] introduced graphical user interface controls enhanced with embedded visualizations. These visualizations are implemented as a "Look and Feel" layer extending the standard Java UI toolkit appearance. However, to implement *Scented Widgets* developers would need to modify the application source code and use that particular UI toolkit. Using the Patina system we can create an application-independent implementation of *Scented Widgets* for standard check and combo boxes. (Figure 16).

For checkbox controls, a small stacked horizontal bar chart is overlaid to the left of the checkbox to indicate the relative

proportion of users who have this option selected. For comboboxes, we show a horizontal bar chart showing the relative frequency of the items which have been selected from the dropdown. The user's currently selected value is shown in a darker shade. For both controls, hovering over the charts shows a larger version with labels. To minimize visual distraction we only display the small visual scent indicators for UI close to the cursor position.

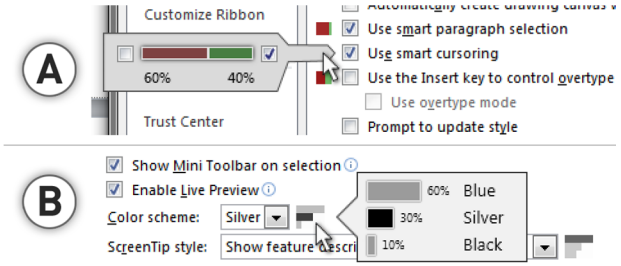


Figure 16. Scented interface for check box (A) and combo box (B) controls.

USHER

The *USHER* system [9] by Chen *et al.* is designed to improve the accuracy of form filling information by learning a probabilistic model of the dependencies between options. Based on this model, the *USHER* system augments the user interface to promote correct user input and alert the user of entries which may be incorrect.

Besides form filling applications, we believe a system like *USHER* could be useful in situations such as preference dialogs where there are often many settings that a user can modify and difficulties can arise if any of them are set incorrectly. We prototyped this by placing a warning icon beside options which may be set incorrectly based on the behavior of other users. We looked at settings individually, but the accessibility data used by the Patina system would allow for creating a probabilistic model for determining outliers in a similar way as is done in the *USHER* system.

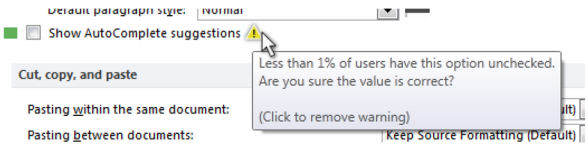


Figure 17. Warning icon and message for setting which are possibly set outside of normal bounds.

CommunityCommands

The *CommunityCommands* system [17, 19] is a recommender system for commands within an application. The active user's usage history is compared to the usage patterns of others in the community, and a list of commands are presented which might be useful to the user. The *CommunityCommands* system relies on in-product instrumentation to collect the usage data, but we are able to provide similar functionality using the data collected from the Patina system (Figure 18).

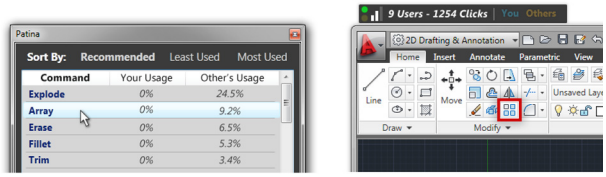


Figure 18. Command recommendation interface (left). Highlighted command in AutoCAD (right).

A list of the UI elements available in the application are presented in a list view, along with how often they are used by the active user and by the community. The list can be sorted to show the commands which the active user uses the most/least, or to present a list of recommended commands which is calculated by finding commands which the community uses a lot, but the active user does not use at all. Advanced collaborative filtering algorithms could be used to generate more robust recommendations.

When the users clicks an item in the list, a rectangular highlight is drawn over the element in the main interface and if the accessibility information includes hotkey data, we can automatically execute the command.

INTERNAL DEPLOYMENT

To get initial feedback of the Patina system, we conducted a short-term deployment evaluation of the system. Because Patina is still a prototype system that needs to run at all times, and collects potentially sensitive data (such as document names) the study was run internally. Eight participants within our organization ran the Patina prototype for 1 week on their office machines while performing their daily computing tasks. To reduce the system load and the amount of data being transferred, the system was modified to only collect data when Microsoft Word was the foreground application.

Usage Data and Feedback

During the deployment the Patina system recorded 8,742 total click events from the eight users. The heatmap overlay was activated a total of 285 times: 92 for the personal overlay, 130 for the community overlay, and 63 times using the combined data. Looking at the area of the regions that the click events occurred in (Figure 19) we can see that 12% of click events were below our 4,096 px^2 threshold where we display the clicks at full intensity, and 80% were above the 90,000 px^2 threshold for events we assume took place in a main canvas area. Since we render these large-area points very transparently because we don't believe they have much informational value, in the future we could consider ignoring those data points completely when they are collected to reduce the data transfer and rendering costs.

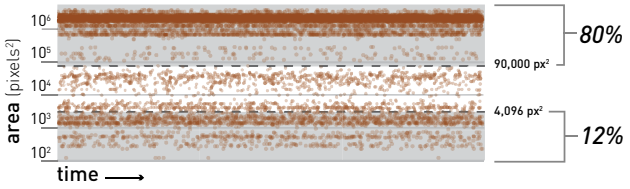


Figure 19. Scatter plot of the region sizes where clicks occurred during the internal deployment.

While using the system the users reported exploring more of the interface than they usually would because they were curious to see what parts of the interface the others were using. Several users discovered interface widgets they were previously unaware of through the heatmap visualization. For example, one user discovered the "zoom slider" in the bottom right corner of the window, and has subsequently adopted the use of that slider for zooming his documents. Another user mentioned that seeing the heatmap from the other users made him realize that he uses a much smaller set of tools than his colleagures. Several of the particiapants mentioned liking the transient overlay, particularly that it would appear when a new dialog box was activated.

PRIMARY USAGE SCENARIOS

Based on the results and feedback from our evaluation, and in addition to our own experiences with the system, we see three primary ways in which the Patina system can be beneficial to users.

Familiarizing New Users with an Application's Interface

New users of complex applications can often be overwhelmed by the number of user interface elements presented on the screen. This can also be a problem for users experienced in one facet of the program when they start exploring a new area of functionality. The Patina system helps in these cases by highlighting areas of functionality which are frequently used, and potentially, the most relevant to begin exploring.

Exposing Single Document Usage Patterns

The relevant application interface elements may be highly dependent on the current working document. By filtering on a per-document basis, users can quickly locate commonly used widgets for that document, or obtain an understanding of what commands and settings other users have used to create or modify a specific document, in collaborative situations.

Continuous Learning and Reflection

For experienced users of an application, the Patina system supports continuous learning by highlighting interface elements which others are using, which could lead to a better overall understanding of the type of task others perform with the software. Personal reflection is supported by highlighting those elements most often used by the active user and comparing that set with the community usage data, providing a way for the user to notice patterns in their own usage behavior that they would otherwise be unaware of.

DISCUSSION AND FUTURE WORK

We have developed an application independent system to collect and display historical usage information within the context of a software user interface. An internal deployment provided some initial insights into the nature of the data that would be collected.

While this internal deployment was valuable at our current stage of research, an important next step will be to perform more formal evaluations. It would be interesting to study how the system would be received and used in a larger scale external deployment. Further designs may need to be considered to handle data from a larger user base. Focused laboratory studies could also be used to evaluate aspects of the design space presented in this paper. For example, we could evaluate the differences between various visual schemes, such as the light and dark background heatmaps, or compare organic heatmap shapes to a rectangular highlighting technique. We could also study the effectiveness of the color schemes used for the heatmaps.

In terms of gerneralizability, one limitation of our work is that it does depend on the accessibility data for an optimal experience. In the absence of such data, it could be useful to explore augmenting our system with pixel-based image analysis techniques. Projects such as Sikuli [7, 8], Prefab [10, 11], and Hurst et al.'s automatic target identification system [15], are all impressive demonstrations of how vision can be used to interpret interface layout and usage and could be used in concert with *Patina* to recognize UI widgets without sufficient accessibility information present.

Future Work

We have only begun to explore how usage metrics can be displayed within the context of software application user interfaces. There are still a number of interesting design opportunities that could be topics of future work.

One important topic which we have not explored is the dependency of usage information between elements in the interface. Similar to how *USHER* learns dependencies between data entry fields [9], Patina could be extended to learn dependencies between user interface parameter values and options. The heatmaps and scented widget values could be updated to show most likely options to be used based on a user's current context.

Alternatively, when working in a preference or configuration dialog, the Patina system could provide a mechanism to view or restore previous states of the entire dialog. This could allow users to quickly review how combinations of parameters have been used in the past, either from their own use, or by other users on their team or from the community.

Another way user interface dependencies could be used is to incorporate command recommender system technology into Patina [17, 19]. A user's usage patterns could be compared to the community's, and the *other user* heatmaps could be generated from the most similar users. Heatmaps could also be used to show the next most likely elements a user will click on, based on their past sequence of interactions. This could guide users through a correct workflow when setting up multiple parameters in a dialog.

Patina could also be bundled with tutorials to help establish which tools in the interface are used to complete the tutorial task. This would be similar to the AdaptableGIMP project [18], which provides custom tool pallets for individual tutorials, but with Patina the layout of the interface would not need to be changed.

Another domain of usage we have not explored is webpage navigation. Typical webpages are composed of rectangular

components [25], similar to that of graphical user interfaces. Such data can be accessed through accessibility APIs through some browsers, or through their Document Object Model. Patina could potentially be used to track and show usage information of a website without integration of special tracking software [29].

Finally, Patina currently visualizes only the usage information of mouse clicks. Because accessibility information does often contain keyboard hotkey associations, hotkey usage information could also be collected and overlaid on the associated icons.

CONCLUSION

The complexity of today's graphical user interfaces exposes users to large information spaces they must navigate to use the software efficiently. The *Patina* system can aid this process by visualizing usage information in the context of the associated user interface elements. Our application independent implementation allows such information to be collected and generated for any application that provides accessibility data, without instrumentation or modification of the actual application. We believe our data collection and interactive UI overlay approach will be useful for future work in application independent desktop services.

REFERENCES

1. Alexander, J., Cockburn, A., and Lobb, R. (2008). AppMonitor: a tool for recording user actions in unmodified Windows applications. *Behavior Research Methods 40*, 413-421.
2. Alexander, J., Cockburn, A., Fitchett, S., Gutwin, C., and Greenberg, S. (2009). Revisiting read wear: analysis, design, and evaluation of a footprints scrollbar. *ACM CHI*, 1665-1674.
3. Bannert, M. (2000). The effects of training wheels and self-learning materials in software training. *Journal of Computer Assisted Learning 16*, 336-346.
4. Baudisch, P., Tan, D., Collomb, M., Robbins, D., Hinckley, K., Agrawala, M., Zhao, S., and Ramos, G. (2006). Phosphor: explaining transitions in the user interface using afterglow effects. *ACM UIST*, 169-178.
5. Bezerianos, A., Dragicevic, P., and Balakrishnan, R. (2006). Mnemonic Rendering: An Image-Based Approach for Exposing Hidden Changes in Dynamic Displays. *ACM UIST*, 159-168.
6. Carroll, J. M., and Carrithers, C. (1984). Training wheels in a user interface. *Comm. ACM 27*, 800-806.
7. Chang, T.-H., Yeh, T., and Miller, R. (2011). Associating the Visual Representation of User Interfaces with their Internal Structures and Metadata. *ACM UIST*, 245-256.
8. Chang, T.-H., Yeh, T., and Miller, R. C. (2010). GUI testing using computer vision. *ACM CHI*, 1535-1544.
9. Chen, K., Hellerstein, J., S., and Parikh, T. S. (2010). Designing Adaptive Feedback for Improving Data Entry *ACM UIST*, 239-248.
10. Dixon, M., and Fogarty, J. (2010). Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. *ACM CHI*, 1525-1534.
11. Dixon, M., Leventhal, D., and Fogarty, J. (2011). Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure. *CHI*, 969-978.
12. Findlater, L., and McGrenere, J. (2004). A comparison of static, adaptive, and adaptable menus. *CHI*, 89-96.
13. Findlater, L., Moffatt, K., Mcgrenere, J., and Dawson, J. (2009). Ephemeral Adaptation: The Use of Gradual Onset to Improve Menu Selection Performance. *ACM CHI*, 1655-1664.
14. Hill, W. C., Hollan, J. D., Wroblewski, D., and McCandless, T. (1992). Edit wear and read wear. *ACM CHI*, 3-9.
15. Hurst, A., Hudson, S. E., and Mankoff, J. (2010). Automatically Identifying Targets Users Interact with During Real World Tasks. *ACM IUI*, 11–20.
16. Hurst, A., Mankoff, J., Dey, A. K., and Hudson, S. E. (2007). Dirty desktops: using a patina of magnetic mouse dust to make common interactor targets easier to select. *ACM UIST,* 183-186.
17. Li, W., Matejka, J., Gossman, T., Konstan, J.A., and Fitzmaurice, G. (2011). Design and Evaluation of a Command Recommendation System for Software Applications. *ACM TOCHI*.
18. Lafreniere, B., Bunt, A., Lount, M., Krynicki, F., and Terry, M. (2011). AdaptableGIMP: designing a socially-adaptable interface. *UIST Adjunct*, 89-90.
19. Matejka, J., Grossman, T., and Fitzmaurice, G. (2011). IP-QAT: In-Product Questions, Answers, & Tips. *ACM UIST,* 175-184.
20. Matejka, J., Li, W., Grossman, T., and Fitzmaurice, G. (2009). CommunityCommands: command recommendations for software applications. *ACM UIST,* 193-202.
21. Nakamura, T., and Igarashi, T. (2008). An application-independent system for visualizing user operation history. *ACM UIST*, 23-32.
22. Pirolli, P., and Card, S. (1999). Information Foraging. *Psychological Review 106*, 643-675.
23. Shneiderman, B. (2003). Promoting universal usability with multi-layer interface design. *CUU*, 1-8.
24. Stuerzlinger, W., Chapuis, O., Phillips, D., and Roussel, N. (2006). User interface facades: towards fully adaptable user interfaces. *ACM UIST*, 309-318.
25. Talton, J. O., and Klemmer, S. R. (2011). Bricolage: Example-Based Retargeting for Web Design. *ACM CHI*, 2197-2206.
26. Terry, M., Kay, M., Vugt, B. V., Slack, B., and Park, T. (2008). ingimp: Introducing Instrumentation to an End-User Open Source Application. *ACM CHI*, 607-616.
27. Willett, W., Heer, J., and Agrawala, M. (2007). Scented Widgets: Improving Navigation Cues with Embedded Visualizations. *IEEE Transactions on Visualization and Computer Graphics 13*, 1129-1136.
28. Wooding, D.S. (2002). Fixation Maps: quantifying eye-movement traces. *ETRA*. 31-36.
29. CrazyEgg. *http://www.crazyegg.com/* (Sept 2012).
30. Mozilla Heatmap. *https://heatmap.mozillalabs.com/* (Sept 2012