Pixel-Based Reverse Engineering of Graphical Interfaces

Morgan Emory Dixon

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington
2015

Reading Committee:
James Fogarty, Chair
Jeffrey Heer
Daniel S. Weld

Program Authorized to Offer Degree:

Computer Science & Engineering

University of Washington

Abstract

Pixel-Based Reverse Engineering of Graphical Interfaces

Morgan Emory Dixon

Chair of the Supervisory Committee:
James Fogarty
Computer Science & Engineering

User interface software is often large, complex, rigid, and difficult to implement. In fact, most of the development time and effort in a typical application is dedicated to the user interface. Not only do interfaces require the programmer to deal with complex input and output, asynchrony, and rapid feedback to the user that is context-dependent, but they also need to run on many different devices and serve a wide variety of people with complex needs. For example, applications may be required to run on the desktop, on mobile touchscreen devices, both for people with motor impairments and able-bodied users, and in multiple languages. As a result, most of today's interfaces require an immense amount of resources to address the increasingly complex ecosystem of devices and users.

This dissertation focuses on *unlocking existing user interfaces* to overcome these challenges. Similar to how Wikipedia opened encyclopedia articles for anyone to edit, we envision a similar democratization of user interfaces, where *anybody can modify the graphical interface of any application*. Any single company or group of developers cannot single-handedly address people's wide variety of needs, but unlocking existing interfaces could diffuse the cost and effort across many people. As a first step towards realizing this vision, this dissertation explores how to modify existing graphical interfaces by reverse engineering their pixels. Working from the pixels enables developers and end-users to modify existing interfaces without compliance from the underlying application. For example, we use our pixel-based methods to implement many different modifications including accessibility enhancements, improved input on mobile devices, interface translation for improved localization, and end-user customization. Researchers have explored pixel-based reverse engineering for decades, but this dissertation presents the first real-time methods for interpreting an interface's pixels and using that reverse engineered structure to modify an application's apparent behavior.

Table of Contents

List of Figures

Acknowledgements

I would first like to thank my advisor and doppelganger James Fogarty. “I both work with and look like James.” James taught me how to focus deeply on a hard problem, how to transform my disorganized thoughts into an elegant framework, and how to separate concerns. The latter was an engineering trick, but James could also apply it to his own work-life balance. As someone who is prone to worrying, this was a valuable skill for me to adopt. James supported me for six years and I am grateful.

I’d like to thank my dissertation committee members for their advice throughout my PhD. Jake Wobbrock boasted about me to others with great confidence. It is much easier to believe in yourself when someone else believes so strongly in you. Dan Weld asked targeted and insightful questions throughout my PhD. David McDonald offered a practical and unique perspective on my work. Finally, Jeff Heer held me to world-class standards while still being supportive and enthusiastic.

I am also grateful to the many other mentors who supported me. I am privileged to be an honorary James Landay student. Landay is the mob boss of HCI, and I feel like the godfather invited me into his home to celebrate his niece’s wedding. Jeff Nichols encouraged me to pick hard problems while still giving myself time to play (and drink fancy cocktails). Desney Tan and Dan Morris were the most enthusiastic mentors I’ve had. Research with them was like jumping into an ongoing pickup baseball game with very few rules. Scott Saponas was a lab mate and research advisor with an unwavering and pure heart. He believed in me strongly. Finally, Ken Hinckley was the nicest and most humble researcher who worked with me.

Many senior students guided me through my first few years in graduate school. I have great memories traveling to conferences and laughing hard with Kayur Patel, who is a grown-up kid like me. Saleema Amershi was supportive and balanced (and she kept Kayur in check). I also received support from Kate Everitt, Susumu Harada, Jon Froehlich, and many others. I’d like to thank Travis Kriplean for being a great confidant, for admitting me to the Invisible College of Hawaii, and for introducing me to the Talking Heads. Finally, Michael Toomim was my most caring friend and a fellow misfit. He encouraged me to be creative, challenged me to question my beliefs, and demonstrated how to escape from of the seemingly important but often-unnecessary rules and obligations we impose on ourselves.

I also had the opportunity to see new students enter the field. The undergraduates who put up with me the longest and taught me the most about mentoring were Stephen Jonany, Orkhan Muradov, and Cullen Walsh. I was also a co-instructor with the wildly creative and enthusiastic younger Fogarty student, Katie Kuksenok. I have fond memories talking and brainstorming with lab mates Shiri Azenkot and Lydia

Chilton. Conrad Nied and Daniel Leventhal were super-enthusiastic collaborators who gave me a push start when my research stalled. Most recently, my research buddy and band mate Katie O'Leary has bravely and passionately dived into a scary new research field with me.

Many thanks also go out to my other equally-rad band mates Michelle Fellows, Courtney Steitz, and Erin Murphy. I have many unforgettable experiences practicing, recording, and playing shows, including one of our first performances at my wedding! They gave me their friendship and helped me grow as a musician and gain confidence in myself.

Thank you to Alan and Inger Osberg, both for a generous scholarship and for many precious dinner parties with their family.

I want to thank my friends Janara Christensen, Tony Fader, Abe Friesen, and Brandon Lucia. Janara was one of the kindest and most dedicated friends I've had. Some of my most hilarious moments were with Tony and Brandon. Finally, Abe helped me survive the last stretch of my PhD with snarky banter, nerdy debates, and bowls of noodle soup.

I'd like to thank my family. Many thanks to my mom for her compassion and support. She encouraged me to pursue my dreams and live true to myself. Thanks to my dad and stepmom Bud and Brenda for challenging me to think critically. Thanks to my sister for her constant care and for inspiring me to be self-disciplined, focused, and organized. Thanks to her husband Ben for commiserating with me during his time as a PhD student. Thanks to my niece Matilda for her phone calls and goofiness. I want to thank my wife's family Jeff, Ruth, Jennifer, Jeremy, Emily, and Ethan for their endless hospitality and support. I could not have made it here without them.

Finally, I'd to thank my wife and best friend Leianna Dixon. This would have been impossible without her enduring love. Her sincerity and kindness inspire me every day.

This work was supported in part by the National Science Foundation under awards IIS-1053868 and IIS-0811063, by Intel, by the UW CSE Hacherl Graduate Fellowship, by a fellowship from the Seattle Chapter of the ARCS Foundation, by a Microsoft Research Award, by an internship at Microsoft Research, by an internship at IBM Research, by the UW CoE Osberg Fellowship, by the Microsoft Endowed University Fellowship, and by the author's wonderful colleagues, family, and friends.

Dedication

To my mom, my best pal in the world.

Chapter 1 | **INTRODUCTION**

User interface software has become a victim of its own success. Graphical event-driven systems are the dominant user interface for computational devices, realized via popular user interface toolkits. However, this success has induced a lock-in effect that demonstrably stifles innovation. For example, human-computer interaction researchers regularly propose new interaction techniques for desktop and mobile devices, enhancements for people with disabilities, better support for localization, and new collaboration facilities. However, these techniques rarely make it outside the laboratory because existing software tools cannot support them. Changing the ecosystem would require a mass adoption of new tools, and most of the applications we use today would need to be rewritten.

This dissertation focuses on *unlocking existing user interface tools* so that developers can explore new ways to build interfaces inside the current ecosystem. For example, a human-computer interaction researcher developing a new interaction technique might evaluate it in several real-world applications (e.g., Adobe Photoshop, Apple iTunes, Microsoft Office). A practitioner or hobbyist who sees the researcher's prototype might then add the technique to several of their favorite applications. Web communities might organize around causes, such as translating interfaces into new languages, improving the accessibility of applications, or updating interfaces to better support ink, gestures, speech, and other advanced interactions. Ultimately we envision a transformative democratization of human-computer interaction, enabled by new tools that unlock existing interfaces.

Realizing this vision is difficult because of the *rigidity* and *fragmentation* of current interfaces. Researchers and practitioners who develop new ideas and techniques generally find it difficult or impossible to add their ideas to complex existing applications. In addition, people generally use a wide variety of applications implemented with multiple underlying toolkits. Adding flexibility to any one application or toolkit is therefore insufficient for techniques that need to work across an entire desktop.

My dissertation addresses rigidity and fragmentation by building upon the single largest commonality of existing graphical interfaces: they ultimately consist of pixels painted to a display. Building from this universal representation, we explore pixel-based interpretation to modify interfaces without their source code and independent of their underlying toolkit implementation. Specifically, this document describes our work on the *Prefab* system, which examines Pixel-Based Reverse Engineering for Advanced Behaviors, and builds upon five fundamental insights:

- Unlocking existing tools requires *real-time* analysis of interfaces. Existing pixel-based methods are not designed to rapidly interpret the entire contents of an interface and are therefore a poor fit for our problem [3,74,76]. For example, Yeh et al.'s Sikuli system uses pixel-based methods for goals like automating repetitious tasks, but the methods require a reported 200msec to identify all occurrences of a single element [74]. Instead, we design our methods to identify all occurrences of many widgets many times per second.
- The graphical desktop is not a physical scene. Computer vision algorithms developed to address such problems as perspective, distortion, shadows, and occlusion may be overkill or even inappropriate for this problem. We instead need algorithms based in an understanding of how interface toolkits render interface elements.
- The internal consistency of an application is important for usability, so a widget's pixels are typically consistent across invocations of an application on the same or different computers. If a system can be taught the definition of a particular widget, it will rarely change.
- Applications are also designed to be externally consistent for usability, and so widgets of the same type are typically illustrated using similar pixels. It is likely not necessary to individually define each widget in every application, but may instead be possible to learn definitions of entire families of widgets (e.g., all Microsoft Windows Vista Steel buttons, all Java Metal checkboxes, all Apple Cocoa scrollbars).
- The commonalities between widgets are also an artifact of how toolkits are implemented. For example, Hudson and Tanaka propose toolkit methods for painting highly stylized widgets using abstract templates that can be populated with specific pixel values [35]. Turning these skinning methods on their heads, it may be possible to recover the templates from example images. We could therefore inform our methods with the workings of toolkits to build more robust and accurate models. However, our methods can simultaneously remain independent of a toolkit's implementation because we are ultimately examining pixels captured from screenshots.

In short, graphical desktops are not physical scenes, but are instead made of pre-fabricated units combined according to very particular rules. Prefab uses raw pixels to reverse engineer interface structure by identifying these pre-fabricated units and then modeling their relations. The next section presents some of the demonstrations we have implemented using Prefab's pixel-based methods. The interaction techniques themselves are taken from the HCI research literature [6,23,28,44,47,67,73], but were previously difficult or impossible to implement in real interfaces. We have applied the techniques to a variety of applications based on different toolkits running on different operating systems (e.g., Adobe Photoshop, Apple iTunes, Google Chrome, Microsoft Office, Mozilla Firefox, Skype, YouTube). All are

implemented entirely based upon reverse engineering pixels, without knowledge of the underlying toolkit or implementation. Real-time responsiveness of our initial methods can be seen in videos recorded on a typical desktop: http://prefab.github.io. This document explores Prefab to demonstrate the thesis:

> **Pixel-based methods for reverse engineering interfaces enable innovation in human-computer interaction by circumventing the rigidity and fragmentation of existing graphical interfaces.**

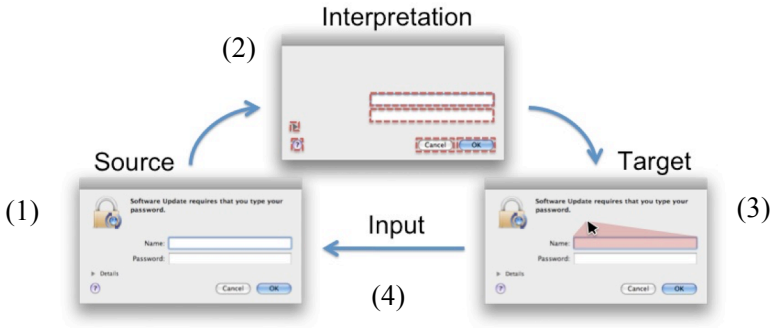
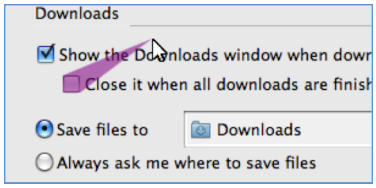


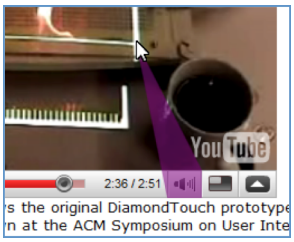
Figure 1.1: Redirection mechanisms allow modification of the apparent behavior or interfaces through modification of their input and output. Prefab enables this with a unique approach to rapid pixel-based interpretation of interfaces.

1.1 Input and Output Redirection

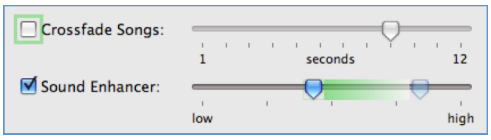
Prefab modifies the apparent behavior of existing interfaces using pixel-based interpretation with input and output redirection. Figure 1.1 illustrates a basic mechanism, where (1) a source window bitmap is captured, (2) the source image is interpreted, (3) the modified interface is presented in a target window (with the source potentially hidden using virtual desktop methods), and (4) input in the target is mapped back to the source, which generates new output that is captured and used to update the target. We discuss prior redirection research in Chapter 2 [38,64,66], but Prefab is the first system to combine pixel-based methods with input and output redirection, allowing it to modify an interface independent of that interface's implementation. The types of modifications that are possible with these mechanisms depend upon the completeness of the available interpretation, so this dissertation focuses on methods for identifying widgets many times per second. However, our pixel-based methods also enable new approaches to intelligently rendering the target window and redirecting input, and so Chapter 6 presents some initial methods for output redirection.

Supporting real-time identification is a key challenge for Prefab. Complex statistical approaches are likely too expensive or inaccurate for the needs of a typical modification. On the other hand, purely manual code-based systems are likely too rigid or brittle to model the wide variety of interface elements. Prefab obtains tractable and scalable real-time identification by combining the advantages of these two approaches. Specifically, we rely on *exact-matching* of raw pixel values to identify widgets in real-time, but we *automatically* determine which pixels to consider according to extracted example images of widgets. This approach has inherent tradeoffs, which we discuss throughout this dissertation. Our related work section also provides a discussion and comparison of other pixel-based approaches as further rationale for our specific design decisions. Ultimately, Prefab is the first system for interpreting an entire interface at frame rate, which enables new classes of modifications to existing interfaces independent of their underlying implementation.

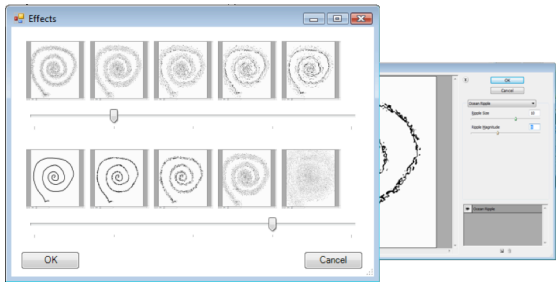




Grossman and Balakrishnan's Bubble Cursor is an important target-aware pointing technique [28], but is difficult to deploy because existing toolkits do not support it. These are screenshots of a Bubble Cursor implemented using Prefab, highlighting the nearest target in a Firefox settings dialog on Macintosh OS X and in a YouTube movie player.



Baudisch *et al.*'s Phosphor uses afterglows to illustrate interface changes [6]. This screenshot of our implementation of Phosphor shows a recently unchecked checkbox and a recently manipulated slider in an iTunes settings dialog on an Apple Macintosh.



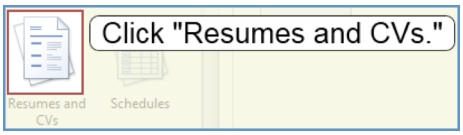
Terry and Mynatt show that Side Views parameter spectrums can support effective exploration of multi-parameter spaces [67]. We use Prefab to create a parameter spectrum for an Adobe Photoshop filter running on Microsoft Windows. We populate the spectrum by using Prefab to automatically interpret the interface of Photoshop's filter dialog.

Figure 1.2: Prefab reverse engineers raw pixels to recover the structure of graphical user interfaces. Because Prefab is agnostic of an application's underlying implementation, it enables advanced behaviors across a wide variety of applications and toolkits. All of these demonstrations are discussed in greater detail in this dissertation and in our associated videos.

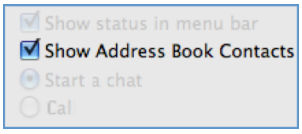
1.2 Chapter Overview

Prefab is a powerful tool that enables the modification of existing applications without compliance from the original developers. With it we were able to re-implement a variety of interaction techniques proposed in the human computer interaction literature that were previously impossible to implement in practice. We examined these techniques in real applications for the first time, and uncovered important insights about their design. This dissertation therefore presents two major classes of contributions: (1) new technical methods that enable modification of existing applications, and (2) insight into the design of interaction techniques enabled by our methods.

Chapter 2 starts with an overview of related work. Our related work first discusses two specific classes of enhancements that were previously considered impossible to implement in practice, but Prefab enables their implementation across a wide variety of real-world interfaces. We then describe prior work on systems for modifying interfaces at runtime and prior pixel-based methods. Chapters 3 through 7 then present our contributions to pixel-based methods along with demonstrative applications, several of which are techniques from the HCI literature described in our related work. Chapters are organized to build upon each other, where the methods and insights presented in one chapter directly address the limitations unearthed when exploring the demonstrative applications presented in the previous chapter. Chapter 7 then concludes our presentation of technical contributions with a study that examines how developers use

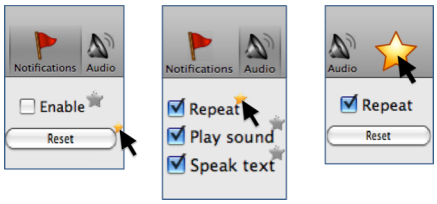


Kelleher and Pausch present the use of stencils to provide tutorials directly within the context of an interface [44]. This is a screenshot of our pixel-based implementation of a tutorial for downloading résumé templates in Microsoft Office.



Findlater *et al.*'s Ephemeral Adaptation improves targeting performance by reducing visual search while maintaining spatial consistency [23]. Likely targets appear as normal, but unlikely targets are initially missing and slowly fade in. This is a screenshot from our implementation of the technique in the context of a Skype settings dialog box running on Mac OS X.

We present a GUI translator that works solely from an interface's pixels. Our translator identifies, interprets, and translates textual content while maintaining the same look and feel as the original application. Here is a portion of a Google Chrome Options dialog running in Microsoft Windows, translated from English into French.



Our methods enable end-user customization of everyday interfaces. We implemented a technique that allows end-users to aggregate commonly used widgets of a tab control into a "favorites" tab. This is shown here in the context of the Skype settings dialog. Clicking on stars (left) adds the corresponding widgets to the "favorites" tab (right).

Figure 1.3: We introduce new pixel-based methods to reverse engineer interface content and hierarchy in real time. These methods enable a new range of advanced behaviors that are agnostic of an application's underlying implementation. All of these enhancements are implemented in the Prefab system and are discussed in greater detail later in this dissertation and our videos available at prefab.github.io.

our tools to implement a state-of-the-art enhancement. Finally, Chapter 8 discusses the remaining limitations of our system and important opportunities for future work.

Chapter 3 presents Prefab's methods for robustly identifying individual widgets from a library of *prototypes,* our representations of widget appearance. Specifically, the methods described in this chapter address three important challenges: 1) decomposing example images of widgets into prototypes, 2) matching the pixels stored in those prototypes against a captured image to identify widgets in real time, and 3) detecting lightweight transitions in a widget's appearance between frames. We validate Chapter 3's methods through a range of demonstrative applications.

Figure 1.2 presents several of these examples, and our videos available at prefab.github.io present them running in real-time. These examples are embedded in a variety of applications implemented using different toolkits running on different platforms. All are implemented entirely based upon reverse engineering pixels, without knowledge of the application's underlying toolkit or implementation details.

Chapter 4 adds the ability to model interface content and hierarchy. We first introduce the use of hierarchy to characterize complex widgets that are composed of multiple elements. We then describe our content regions and how they can be used to recover less predictable interface elements, such as screen-rendered text. Finally we show how content and hierarchy can be combined to recover the

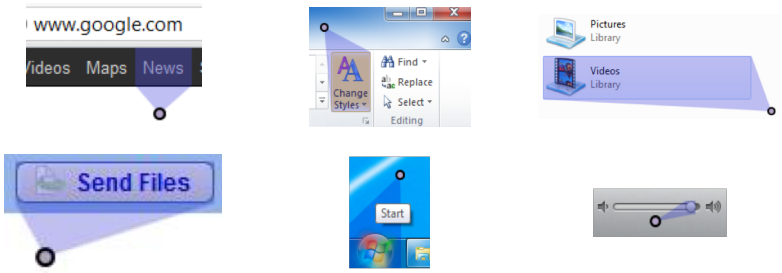


Figure 1.4: Chapter 5 presents a practical implementation of a general-purpose target-aware pointing enhancement. Specifically we implement Grossman and Balakrishnan's Bubble Cursor across the Windows desktop.

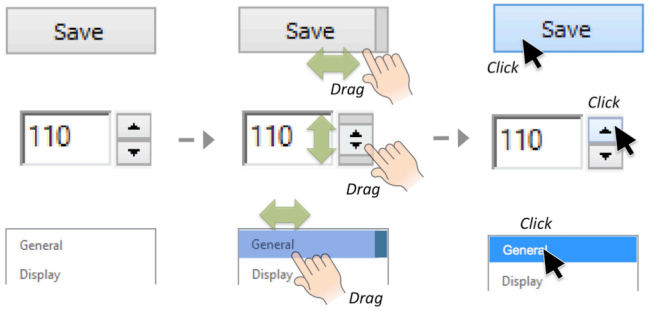


Figure 1.4: Chapter 6 presents new our pixel-based methods to explore Moscovich et al.'s Sliding Widgets in real-world interfaces. We overlay Sliding Widgets throughout Microsoft Windows 8, replacing standard mouse-based elements to improve interaction with hybrid touch-and-mouse devices.

structure of an entire interface in real-time. Figure 1.3 illustrates several applications enabled by these methods. Importantly, these applications demonstrate our ability to distinguish between apparently identical widgets using our recovered hierarchy to compute unique path descriptors for each element.

In Chapter 5, we develop methods for implementing an interaction technique that functions across the entire desktop. We also report on the limitations of this general-purpose enhancement unearthed by examining the technique in the complexity of real-world interfaces. We therefore contribute important progress toward real-world deployment of interaction techniques like these and shed light on the gap between understanding techniques in controlled settings versus behavior with real-world interfaces. Our implementation functions across the desktop, as illustrated in Figure 1.4. The methods we develop can be applied to any modern desktop, and they can be extended to support other target-aware techniques. Specifically, we architect our enhancement to separate *identification* of interface elements from *interpretation* of how to target those elements. We implement identification using our methods from previous chapters. We implement interpretation by annotating interfaces with desired targeting behavior. We also develop interfaces for correcting errors in both levels, and we argue that *social* mechanisms for identification and interpretation are essential to deploying any general-purpose enhancement.

Chapter 6 presents novel methods for modeling widget *state* and *style*. We first describe *state*, our methods for monitoring and manipulating a widget's behavior across frames via state machines. Many

potential enhancements require an understanding of how an interface changes across multiple frames (e.g., if a checkbox becomes checked, if a slider thumb has moved). More advanced enhancements also require manipulating the interface (e.g., sending click events to set a checkbox, dragging a slider). We then describe *style*, our methods for intelligently rendering new widgets based on Prefab's library of prototypes. Capturing an interface's style is important because many runtime modifications do not fully alter the appearance of an existing interface, but instead directly overlay new elements onto the interface. Therefore it is important that these enhancements are styled to preserve consistency with the existing interface. Chapter 6 also combines our methods for state and style in an implementation of Moscovich et al.'s Sliding Widgets, touch widgets activated by sliding a moveable element [47]. Our implementation dynamically overlays Sliding Widgets on mouse-based interface elements throughout Microsoft Windows 8. For example, Figure 1.5 and our associated video present screenshots of our implementation in a variety of popular interfaces (e.g., Microsoft Word 2013, Adobe Reader, Gmail in the Google Chrome Browser, and Windows Explorer).

In Chapter 7 we introduce a developer toolkit for pixel-based enhancements, focused on two areas of support. *Prefab Layers* helps developers write interpretation logic that can be composed, reused, and shared to manage the multi-faceted nature of pixel-based interpretation. *Prefab Annotations* supports robustly annotating interface elements with metadata needed to enable runtime enhancements. Chapters 3 through 6 present important pixel-based methods, but unfortunately these are brittle and difficult to use without additional structure. Together, Prefab Layers and Prefab Annotations help developers overcome subtle but critical dependencies between code and data. We validate our toolkit with (1) demonstrative applications and (2) a lab study that compares how developers build an enhancement using our toolkit versus state-of-the art methods. Our toolkit addresses core challenges faced by developers when building pixel-based enhancements, potentially opening up pixel-based systems to broader adoption.

Finally, Chapter 8 discusses important remaining challenges in pixel-based reverse engineering and describes opportunities for future research. Specifically we (1) describe opportunities for combining pixel-based methods with other strategies for runtime modification, (2) discuss possibilities for more efficient strategies for reverse engineering, and (3) envision community-driven authoring tools that democratize the design and implementation of interfaces. This dissertation represents an important first step towards this future, enabling a practical approach to adding advanced behaviors to new and existing interfaces independent of their underlying interface toolkits.

Chapter 2 | **RELATED WORK**

In this chapter, we review related work involving pixel-based reverse engineering. This chapter is divided into five sections:

- In section 2.1, we overview two important classes of applications enabled by Prefab: *target-aware pointing* and *device diversity*. Target-aware pointing techniques are designed to improve the fundamental task of pointing and clicking by leveraging knowledge of the sizes and locations of clickable targets in an interface. Device diversity enhancements are designed to reduce the burden of developing multiple interfaces for different devices. Many of these techniques have been proposed in the literature, but have been considered difficult or impossible to build in practice. This section motivates and reviews these techniques, and later chapters describe how Prefab enables their implementation in real-world interfaces.
- In Section 2.2, we present traditional approaches to modifying existing interfaces at runtime. These include methods that inspect the accessibility API, toolkit injection systems, and web-based systems. Most runtime modification tools rely heavily on the underlying implementation of interfaces. On the other hand, pixel-based methods are independent of the underlying toolkit implementation. This section describes the tradeoffs between these approaches.
- Section 2.3 describes pixel-based methods developed around the same time or after Prefab. These methods differ from Prefab in two important ways. First, many of these methods are tailored to specific applications. In contrast, Prefab's general-purpose methods are designed to support a wide variety of enhancements. Second, many of these applications do not focus on the real-time modification of interfaces and therefore make important performance tradeoffs. This section discusses these tradeoffs and how the constraint for real-time performance informed the design of our system.
- Section 2.4 describes prior hybrid systems that combine pixel-based methods with other traditional approaches. These include methods that augment the accessibility API with structure recovered from pixel analysis.
- Finally, Section 2.5 describes some related systems that make use of input and output redirection. These systems further motivate our design decisions aimed to enabled real-time performance, because these decisions simultaneously result in models of widgets that can be used to render more sophisticated enhancements.

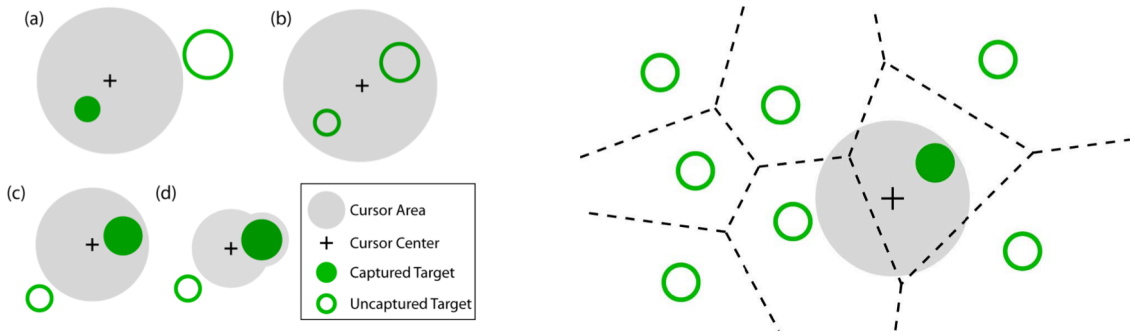


Figure 2.1: Grossman and Balakrishnan's Bubble Cursor expands to ensure that the nearest target is selected [28]. This figure from their original paper shows the cursor in gray, targets in green, and the Voronoi space that defines effective size of each target.

2.1 Enabled Interaction Techniques

Current interface tools have made it difficult or impossible to implement a variety of compelling applications, including accessibility enhancements, interface mashups, interface debugging software, automation tools, in-context tutorials, collaboration software, improved interaction on mobile devices, and localization improvements. We validate the methods proposed in this dissertation by implementing a broad variety of these applications in the context of real-world interfaces. Chapter 5 and Chapter 6 also provide deeper examinations of two classes of enhancements, *target-aware pointing* and *sliding widgets*. This section therefore provides an overview of these techniques that were proposed in the HCI literature.

Target-Aware Pointing

Target-aware pointing is a family of techniques that can improve the fundamental task of pointing and clicking. The defining property of a target-aware pointing technique is that it requires knowledge of the size and locations of targets in an interface. Leveraging this knowledge, target-aware pointing techniques can significantly outperform other pointing facilitation techniques and they ultimately have great potential to improve the efficiency of interaction. As a result, over 20 years of research has explored a wide variety of target-aware pointing techniques that range from general-purpose cursors designed to run across the desktop, to techniques optimized for specific contexts or users. For example, researchers have explored techniques designed to improve accessibility for people with motor impairments and also to improve interaction on touchscreens.

The most efficient target-aware pointing technique to date is Grossman and Balakrishnan's Bubble Cursor [28], an area cursor that dynamically expands to always capture the nearest target. Figure 2.1 shows two abstract visualizations that were presented in the original Bubble Cursor paper. The green circles represent clickable targets in an interface, the gray circle represents the dynamically resizing Bubble Cursor, and the dotted lines to the right visualize the Voronoi space that defines the effective size

of each target. The technique also supports targeting in dense layouts, which is a fundamental limitation of standard area cursors [42]. Because of this advantage, the cursor has been shown to significantly outperform the standard point cursor by 16%.

The Bubble Cursor is designed for average users, but other research has explored alternative enhanced area cursors specifically optimized for people with motor impairments. For example, Findlater et al. explores a breadth of enhanced area cursors, each optimized for different abilities such as cerebral palsy, multiple sclerosis, and Parkinson's [22]. The advantage of each technique results from employing a different strategy for acquiring targets. For example, the Visual-Motor-Magnifier increases both motor and visual space by first resizing targets and then applying the Bubble Cursor technique. In comparison to the Bubble Cursor, these techniques can have a much larger improvements of over 50% for people with motor impairments.

Researchers have also explored a variety of techniques beyond area cursors. One of the earliest examples is Worden et al.'s Sticky Icons [73], a technique designed for elderly people. Specifically, the technique dynamically adjusts the mouse's control-display gain to reduce the effects of jittery hand movements (i.e. the cursor's speed decreases when moving over a target, creating a "sticky" effect). Other similar techniques include gravity wells [39], force fields [1], semantic pointing [8], bubble targets [15], object pointing [30], and drag-and-pop or drag-and-pick [5]. Similar to the enhanced area cursors, each of these techniques offers different strengths that could benefit users with specific abilities or contexts.

Unfortunately, despite the promise of these techniques, few have been deployed or evaluated in real-world interfaces. This is primarily because most existing applications do not expose the sizes and locations of their clickable targets. This chapter's section on runtime modification describes frameworks that attempt to expose the underlying interface elements of an existing application (e.g., the accessibility API). But these frameworks are notoriously incomplete and difficult to correct. Developers who want to implement a target-aware pointing technique are therefore faced with re-implementing an entire application from scratch. As a result, most target-aware techniques have been limited to abstract targets in isolated test beds. We therefore have little understanding of how these techniques behave in real-world interfaces, and the potential of these techniques have largely been unrealized.

The difficulty of deploying target-aware techniques has motivated developers to explore *target-agnostic techniques*, which aim to improve pointing without knowledge of targets. We are aware of only four such techniques: conventional pointer acceleration (cf., [10]), PointAssist [33], the Angle Mouse [72], and the Pointing Magnifier [40]. In addition, Hurst et al.'s use of click history to approximate gravity wells can be

considered a target-agnostic technique [37]. Despite the ingenuity of these techniques, they are inherently limited by their ability to consider only mouse kinematics and clicks. Unfortunately no target-agnostic technique has demonstrated performance superior to the best target-aware techniques.

My dissertation provides a foundation for the future deployment of target-aware techniques. Chapter 3 to Chapter 5 present a variety of target-aware pointing techniques including our general-purpose implementation of the Bubble Cursor that operates across the entire Microsoft Windows desktop. Chapter 5 also reports on limitations of the techniques unearthed by examining the Bubble Cursor in the complexity of real-world interfaces. We therefore contribute important progress toward real-world deployment of an important family of techniques and shed light on the gap between understanding techniques in controlled settings versus behavior with real-world interfaces.

Device Diversity

In roughly ten years, we have seen the emergence of smartphones, tablets, a range of smart appliances, netbooks, wall-sized displays, interactive televisions, advanced game consoles, smart watches, and many new health and fitness devices. Unfortunately this growing device diversity poses a serious challenge for developers because each of these devices requires a unique interface tailored to its particular input and output modalities. Developers are therefore faced with manually implementing interfaces for each of these devices, which can result in a combinatorial explosion of code.

The challenge of supporting device diversity is magnified in the emerging class of *hybrid* devices that support both touch and mouse input, such as the Microsoft Surface tablet and the Lenovo Yoga. Because hybrid devices require multiple interfaces for a *single device*, developers either: (1) provide standard mouse-based interfaces with small and densely arranged targets, or (2) manually implement an entire alternate interface optimized for touch. Developers often abandon the latter due to its difficulty, and leave most interfaces optimized for mouse-based interactions. As a result many applications are completely unusable with the touchscreen, which completely negates the benefits of the hybrid design.

Researchers have therefore explored a variety of approaches to address device diversity, ranging from lightweight interaction techniques that improve input on legacy interfaces to fully automatic generation of new interfaces. Most of the former explores interaction techniques with a focus on the *fat finger problem,* the ambiguity caused when a finger touches multiple targets. For example, Vogel and Baudisch's Shift technique enables fine cursor adjustment with a finger by creating a callout that shows a copy of the occluded screen area and placing it in a non-occluded location [68]. Shift is similar to the fundamental text manipulation technique on every Apple iOS device, demonstrating the potential impact of such

techniques. Other promising techniques that were more specifically designed for interaction with legacy devices include Pointing by Zooming, Rubbing, back-of-the-device interaction, and even see-through devices [2,4,7,70,71].

Although these interaction techniques offer great potential, many of them remain difficult to evaluate and deploy in the context of real interfaces due to the challenges in modifying existing interfaces at runtime. Researchers have therefore investigated automated approaches to generating entirely new interfaces from scratch. For example, Gajos et al.'s SUPPLE automatically generates new interfaces from abstract models based on properties of the device and abilities of its users [26,27]. Unfortunately these models are difficult to author because they preclude direct manipulation of interface appearance, which restrict control from the designer and results in less predictable interface layouts. These models would also require existing interfaces to be completely re-implemented using these tools, which would be difficult and expensive. Chapter 8 describes some opportunities for future research that combines our pixel-based models with SUPPLE's abstract models to automatically retarget existing interfaces.

My dissertation addresses the limitations of both real-world implementation of interaction techniques and automatic interface generation. For example, Chapter 6 explores a real-world implementation of an interaction technique designed to improve hybrid devices. Specifically, we implement Moscovich et al.'s sliding widgets, touch widgets activated by sliding a moveable element [47]. Our implementation dynamically overlays sliding widgets on mouse-based interface elements throughout Microsoft Windows. For example, Figure 1.4 and videos on our website present our implementation in a variety of popular interfaces. We replace elements in the applications using various types of sliding widgets, including sliding buttons, sliding spinners, sliding toggles, and sliding dropdown menus. This implementation can drastically lower the barrier to designing interfaces for hybrid devices because developers implement a single mouse-based interface for each application, which is then transformed to support sliding interactions when the user switches to touch input.

Our implementation of sliding widgets is the most advanced pixel-based enhancement to date, and it serves as an important step towards fully automatic interface generation. For example, Chapter 6's methods for recovering widget state machine models could aid in the recovery of abstract SUPPLE models from *existing interfaces*. Automatic recovery of abstract models could eliminate the need for developers to manually author the models by hand, and circumvent the re-implementation of legacy applications. Automatic recovery could also empower designers rather than restricting them. For example, designers could create example interface layouts using their preferred direct manipulation authoring tools, and then abstract models extracted from these examples could be used to generate alternate candidate

designs tailored for different devices. Prior research has suggested that the presentation of many example interfaces in parallel can dramatically expedite the iterative design process [32], and so we believe this approach could radically transform how designers create new interfaces. Sophisticated applications like these directly motivate this dissertation. Throughout this document we provide methods that serve as important building blocks, and Chapter 8 discusses some of the remaining steps towards such a future.

2.2 Traditional Approaches to Runtime Modification

The problems of rigidity and fragmentation of interface toolkits have been known for decades. Researchers have therefore explored a variety of approaches to *modifying existing interfaces*, rather than requiring developers to rebuild entire interfaces from scratch. As a result, innovations in runtime modification have enabled a range of applications, including interface mash-ups, accessibility enhancements, in-context help for complex interfaces, automatically generated interface tutorials, and automation of repetitious interactions. This section discusses the strengths and limitations of prior work in runtime modification as motivation for pixel-based methods, which are discussed in the following section.

Edwards et al. [21] and Olsen et al. [53] present some of the most classic work on runtime modification [21]. Specifically they modify existing interfaces by replacing the toolkit drawing object and intercepting commands (e.g, draw_string). They use this to update old interfaces with new functionality, such as search and bookmark widgets. Other classic work explores one of the first accessibility systems ever implemented. Specifically Mynatt et al. explores a screen reader system that exposes widgets in the X Windows system and maps those widgets to an audio-based representation [49].

The demand for accessibility enhancements ultimately led to the development of the accessibility API. The accessibility API is specifically designed to expose the locations and content of existing interface elements for accessibility enhancements, such as screen readers or magnifiers, but researchers have also explored alternative and more advanced uses of the accessibility API. For example, Stuerzlinger et al.'s UI Façades system recovers the locations of widgets using the X Windows System's accessibility support in order to support a broad range of enhancements [64]. These enhancements include duplicating widgets for improved control from secondary displays, replacing widgets with simplified controls, and overlaying zooming interaction techniques for efficient image manipulation within existing applications.

More recent examples of runtime modification take a similar approach as Edwards et al. and Olsen et al., but leverage toolkit injection techniques to intercept and replace commands. For example, the Scotty system leverages hooks exposed in Appple's Cocoa Toolkit in order to dynamically load code into program space [20]. The WADE system explores a similar toolkit injection strategy to capture widgets

from the Microsoft .NET toolkit [46]. Specifially, WADE enables modification of .NET applications with a WYSIWYG authoring environment where developers can add, remove, and modify existing widgets.

In contrast to toolkit injection and accessibility API techniques, other work leverages the open nature of a website's Document Object Model (DOM) to access and modify existing interfaces. For example, Fujima et al.'s Clip, Connect, Clone system inspects a webpage's DOM to support three specific customizations: (1) issuing multiple repetitive submissions to the same form, (2) piping results from one application to the input of another, and (3) comparing and choosing amongst the results from multiple requests [24]. Another example of webpage modification is Nichols et al.'s Highlight system [50,51]. Highlight provides an authoring tool tailored for automating repetitive operations, integrating multiple websites, and transforming a site's appearance.

Researchers have also explored more general programming languages to support a broader range of webpage modifications. For example, Bolin et al. present ChickenFoot, an end-user programming system implemented in the browser to support automation and customization of existing web pages [9]. Specifically, ChickenFoot identifies DOM elements by matching patterns generated from keywords provided by the programmer. Programmers can then manipulate the identified elements to automate interactions with the webpage or customize aspects of the page.

ChickenFoot enables individual end-users to modify existing webpages, but there has also been work to support collaborative authoring and sharing of scripts for webpage modification. Specifically, Hartmann et al.'s d.mix provides a tool for creating web mash-ups that automatically generates the underlying service calls that yield the requested page elements to be modified or copied [31]. Importantly, their tool includes a wiki-based hosting environment so the scripts can be shared, modified. These scripts also serve as examples that can be modified to enable rapid experimentation of ideas. Example d.mix applications include rewriting content for mobile devices, ubiquitous computing services for monitoring and lighting control, and automatically extracting web content into a user's calendar.

Unfortunately, all of these runtime enhancements require implementation particular to the underlying toolkit. This dependency poses two important limitations. First, toolkits are frequently incomplete because application developers fail to properly implement the API. For example, Hurst et al. found 25% of widgets are completely missing from the accessibility API [36]. Similar problems with incompleteness manifest on the web when developers implement custom controls (e.g. a video player and its controls, or a custom slider implemented with DIV elements). The severity of this problem is magnified by the fact that most toolkits can be corrected only by an application's original developer (or somebody else with the

underlying source code). The second problem is that the fragmentation of interface toolkits makes it difficult or impossible to implement general-purpose enhancements (e.g. a cursor that runs across the entire desktop). People typically use a variety of applications implemented with several toolkits, so a general-purpose enhancement would require custom methods tailored to each.

2.3 Pixel-Based Methods

In contrast to the runtime modification approaches described in the previous section, pixel-based methods do not require cooperation or additional effort from the developers of the original interface and also circumvent fragmentation of interfaces and toolkits. Leveraging these advantages, researchers have explored a variety of pixel-based approaches to recovering and modifying the structure of an interface. This section discusses and compares the pixel-based systems introduced prior to or around the same time as Prefab, as well as the many pixel-based methods introduced subsequently to our work.

General-Purpose Methods

Pixel-based methods were initially proposed to support research in interface agents and programming by example. One of the earliest examples is the Triggers system by Potter [58], which supports the automation of interfaces based on examples of screen changes. Unfortunately Potter reports that the visual analysis of Triggers is limited to black and white displays and requires roughly 3.5 seconds to identify a the occurrence of a single example image pattern. These original methods demonstrate the initial promise of pixel-based analysis of interfaces, but are clearly insufficient for real-time applications.

Moving beyond Potter's original methods, Zettlemoyer et al. examines pixel-based identification of widgets for their IBOTs and VisMap systems [76,77], and St. Amant et al. provided similar explorations of programming-by-example in SegMan [62,63]. First, the IBOTs work explores interface agents that interpret pixels to automate software logging. Second, VisMap demonstrates more extensive manipulation of existing graphical interfaces with an autonomous solitaire-playing agent and a user-controlled visual scripting program. Finally, SegMan built upon these methods to recover cognitive models to explain and evaluate existing interfaces. Unfortunately, these methods require code-based descriptions of the *appearance* of widgets, making the development of applications tedious and error-prone. For example, Zettlemoyer et al. report that 40% of VisMap code is specific to the particular Microsoft Windows widgets it recognized. SegMan also found their performance insufficient for interactive applications.

More recently Yeh et al. developed Sikuli, a scripting language that supports image-based interface search and automation [74]. The authors demonstrate a variety of tasks programmed using Sikuli, ranging from the automation of repetitious tasks, to macros for closing pop-ups. The authors also use Sikuli's

programming tools to implement a general-purpose contextual help enhancement [75] and a sophisticated UI testing framework [12]. Sikuli uses computer vision methods (template matching and voting based on invariant local features), which are some of the most advanced pixel-based methods used today. Unfortunately, Yeh et al. report that their methods require 200msec to identify all occurrences of a *single* target in an image of a desktop. In contrast, recall that Prefab must identify *many* elements many times per second. This contrast highlights an important performance tradeoff that we consider in our exploration of Prefab. Off-the-shelf computer vision techniques like template matching are designed to identify occurrences of objects invariant to lighting effects and object orientation, so they typically sacrifice precision and speed for *high recall*. Instead we explore methods that rely on exact-matching of raw pixel values because they offer *high precision* and *fast performance*. This approach satisfies our goals for real-time modification because it requires minimal computation, and it can be extremely precise due to the fact that interfaces are procedurally rendered.

Beyond speed improvements, another critical difference between Prefab and the methods presented above is that we provide extensible models that can accurately identify a wide range of interface elements. Specifically, our methods are informed by the extensible ways in which existing toolkits render widgets. These toolkit methods are described in early work by Hudson and Smith [34] and subsequently Hudson and Tanaka [35]. Hudson and Smith first propose toolkit support for separating interface style from content, drawing an analogy to painting the same text with different fonts. This separation allows designers to develop styles at least partially independent of the specific user interface components, similar to how modern CSS styles are implemented independent of HTML content.

Hudson and Tanaka build upon this approach, developing methods for painting highly stylized widgets. Their methods include an eight-part border based in painting fixed corners and variable edges, analogous to the eight-part model we discuss in the next chapter. Prefab turns these approaches to painting widgets on their heads, separating recognition of interface content (described by Prefab models) from style (described by Prefab prototypes). Prefab is simultaneously informed by the workings of user interface toolkits and independent of a toolkit's implementation: our eight-part model can characterize many widgets regardless of whether they were painted using Hudson and Tanaka's eight-part method.

Application-Specific Methods

The pixel-based methods above were designed as general-purpose tools, but other research has explored application-specific methods. For example, one of the earliest pixel-based applications is Olsen et al.'s ScreenCrayons, which supports annotation of documents and visual information in any application [54]. This work builds upon the universality of pixels, allowing users to overlay handwritten notes on top of

any application, but does not interpret the pixels of those images. Other examples include more advanced pixel-based analysis, ranging from off-the-shelf computer vision techniques to highly domain-specific methods tailored to the needs of the specific application.

Several examples explore contextual and video-based tutorials. For example, Banovic et al.'s Waken system interprets and augments captured video tutorials of existing interfaces [3]. Specifically, they examine changes in pixel-values between frames to identify cursor movements and widgets. Their system also extracts templates of elements based on these changes for improved performance. Pongnumkull et al. similarly explore video tutorials, but use pixel-analysis to link the tutorials with running instances of their corresponding applications [57]. The authors of this work report that their methods are similar to those presented in Sikuli, and focus more on problems relevant to working with video tutorials. As such, the methods in both of these applications are tailored for offline analysis of previously-captured video footage and are insufficient for real-time modification.

Similar to video-based tutorials, researchers have explored video-based tools for capturing and navigating document workflow histories [29]. The Chronicle system captures and links video footage of an application in use with the user's content, and then provides navigation tools for revisiting this history. Users can indicate specific content of interest, and see the workflows, tools, and settings needed to reproduce the associated results, or to better understand how the content was created. The pixel-based methods used by Chronicle are primarily focused on linking captured screenshots with application content and interaction events. The authors therefore modify an open-source application in order to obtain the content and intercept the interaction events, leaving relatively straightforward pixel analysis.

In addition to applications that analyze video footage, researchers have also explored new window managers that leverage pixel-based analysis. Specifically, Waldner et al. explore *importance driven* window compositing, which identifies and exposes salient regions of pixels in existing windows [69]. Their system combines existing techniques for measuring visual saliency with compositing techniques to expose important content of the desktop that would otherwise be occluded by other windows.

2.4 Hybrid Methods

The strengths and limitations of pixel-based methods versus toolkit introspection motivate hybrid strategies that combine the two approaches. We are aware of only two hybrid systems. First, Chang et al. explore several synergies in PAX [11]. Specifically they use pixel-level analyses in combination with the accessibility API to locate screen-rendered text, then use Sikuli to find elements in portions of the screen where the accessibility API's representation is incomplete [11]. Second, Hurst et al. use several

pixel-based techniques to improve the accessibility API's detection of target boundaries [36]. Importantly, this dissertation focuses solely on pixel-based methods because the capabilities of any hybrid approach will ultimately depend on the strength of its individual components. Our current work also focuses on pixel-based methods because they can be used to obtain an entire DOM-like structure without cooperation of an underlying application, and future work could extend these to hybrid approaches if needed for more complex or niche interfaces. Finally, these two existing hybrid systems improve the completeness of the accessibility API, but we believe future hybrid systems could also work in the opposite direction to improve aspects pixel-based methods. Chapter 8 discusses potential approaches like this, and offers steps forward for exploring hybrid systems in future work.

2.5 Input and Output Redirection

The main contributions of this dissertation center around pixel-based interpretation. However, Chapter 1 describes how Prefab modifies the apparent behavior of existing interfaces via input and output redirection. To the best of our knowledge we are the first real-time system to combine pixel-based analysis with input and output redirection. In general, most runtime modification systems have not explored sophisticated input and output redirection because they depend heavily on the capabilities of the underlying pixel-based methods.

One extreme example of a system that redirects the output of an application is Tan et al.'s WinCuts, which does not perform any analysis on the application's pixels [66]. Specifically, WinCuts allows subdivision of windows into smaller regions by redirecting their pixels. The authors use this technique to render snippets of applications for cross-device interaction, to enable collaborative presentations with shared screen content, and to organize content on displays with limited screen space.

The most similar use of IO redirection to Prefab is probably Stuerzlinger et al.'s UI Façades [64], which uses hooks exposed by the X Window server and the accessibility API to copy and paste screen regions, send input events to existing applications, and even replace widgets from existing interfaces. However, these methods are fundamentally limited to enhancements that do not require an understanding of an existing application's *appearance*. For example, the accessibility API exposes widget models, but not necessarily their on-screen view (e.g., the pixel coordinates of a slider's thumb are intentionally encapsulated). These methods are therefore incapable of implementing our phosphor enhancement shown in Figure 1.2 and our sliding widgets enhancement shown in Figure 1.4, because these use Prefab's reverse engineering models to render new widgets stylized to match the underlying interface.

Similar to UI Façades, few pixel-based systems have explored sophisticated output redirection techniques. One of the most sophisticated example is Banovic et al.'s intelligent rendering of dropdown menus using reverse engineered structure in Waken [3]. Specifically the authors add the ability to interactively expand and collapse top-level menu contents in existing video tutorials. However, this functionality is limited to tool tips and menus that are recoverable by the particular frame-differencing technique used for widget identification in their enhancement. In contrast we present generalized methods for rendering a wide variety of widgets in a range of styles. We also explore ways to remove content from widgets and expand widgets in order to add additional content.

Chapter 3 presents our initial work that intelligently renders Phosphor widgets, Chapter 4 demonstrates how Prefab can be used to replace widget content and expand widgets, and then Chapter 6 presents our generalized methods for capturing widget style and rendering new widgets. These methods further motivate our decision to explore alternative approaches to off-the-shelf computer vision techniques and other statistical approaches. For example Waken and Sikuli rely on fuzzy matching to identify elements and as a result they cannot render new interface elements based on their models, whereas our methods for rendering new widgets directly stem from our modeling of a widget's exact pixels.

The favoring of exact and deterministic over stochastic and forgiving methods is a major theme of this dissertation. In each chapter we make major design decisions based on this principle. While this introduces important tradeoffs, our methods enable the most sophisticated real-time modifications of existing graphical interfaces. Our methods are also general and can be used to interpret a wide variety of enhancements implemented in a range of toolkits.

Chapter 3 | **PROTOTYPE-BASED IDENTIFICATION**

Two fundamental challenges in pixel-based methods are (1) efficiently and reliably identifying interface elements from captured screenshots and (2) using the reverse engineered structure to implement advanced behaviors like those in Figure 1. This chapter first introduces each of Prefab's major components and describes their relationships. We then provide an overview of these components, as well as two Prefab models that we use as examples throughout the chapter. Next we describe the main portion of our technical content, including how Prefab uses a library of prototypes to reverse engineer a graphical interface. We then validate Prefab's core methods in three example applications. Finally, we discuss the methods presented in this chapter, their limitations, and opportunities that will be addressed in the following chapters and in future work.

The specific contributions of this chapter are:

- Real-time methods for pixel-based reverse engineering of interface structure. Informed by how user interface toolkits paint interfaces, these methods feature a separation of the modeling of widget layout from the recognition of widget appearance.
- Initial methods supporting the creation of Prefab prototype libraries, including a branch-and-bound method for fitting prototype parameters according to positive and negative examples of occurrences.
- Validation of these methods through an exploration and discussion of three demonstrative enhancements implemented in the context of real interfaces.
- A discussion of these core methods, their limitations, and opportunities for future work.

3.1 Real-Time Identification Methods

The major components of Prefab's architecture are: *models*, *prototypes*, *parts*, *constraints*, and *transitions*. This section briefly introduces each component so that future sections can provide a detailed discussion of their usage and relationships.

A *model* consists of a set of abstract *parts* and a set of concrete *constraints* regarding those *parts*. For example, a typical *model* might include several *constraints* requiring that particular *parts* are adjacent. The *parts* of a *model* are abstract, and so a *model* does not describe any particular widget or set of widgets. Instead, a *model* describes a pattern for composing a set of *parts* to create a widget.

Parts can be either *features* or *regions*. A *feature* stores an exact patch of pixels (i.e., exact colors in a spatial arrangement of an exact size). Every *model* includes at least one *feature*, as will be important when we discuss how Prefab identifies elements in a particular interface. A *region* stores a procedural definition of a method for generating a set of pixels in an area of variable size (e.g., painting a repeating pattern, painting a gradient). Because the same *parts* can be arranged in many different ways, they alone do not describe any particular widget or set of widgets.

A *prototype* parameterizes a *model* with concrete *parts*, characterizing both the appearance of a set of *parts* and applicable *constraints* upon the relationships of those parts. A *prototype* therefore describes the appearance of a particular widget or set of widgets (e.g., the Mozilla Firefox Home toolbar button, all Microsoft Windows Steel buttons). Prefab is implemented as a library of *prototypes*, together with methods for effectively applying those prototypes to identify *occurrences* of widgets.

Prefab separates widget layout from its appearance (i.e., the abstract *parts* of a *model* are specified to create a *prototype*). This separation is critical to Prefab and is informed by how toolkits paint widgets. For example, toolkits often use a pre-specified layout to support widgets with content of varying sizes. A common case is painting a border of the required size and then the content of the widget within that border (e.g., centering a label or an icon within a button, displaying the current value of an editable text field). Similarly, a slider widget fundamentally consists of a trough and a thumb, though different toolkits may paint these elements with different appearances. Prefab separates such general insight about how widgets are drawn from details of the appearance of individual widgets (e.g., the stroke used to paint a border, the shape of a slider's thumb). For example, a *model* of eight *parts* creating a rectangular border is capable of describing a wide variety of widgets.

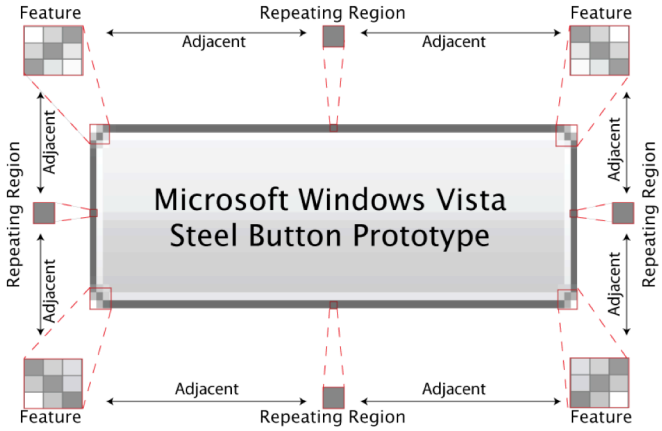


Figure 3.1: This Prefab prototype for Microsoft Windows Steel buttons is an example of an eight-part model. Four features define the corners, each edge is defined by a region, and constraints require the parts form a rectangle. This prototype recognizes all Microsoft Windows Vista Steel buttons, independent of their interior content.

Prefab's final major component is a *transition*. Although many potential applications of Prefab can be based on reverse engineering a single frame or state of an interface, many others require interpreting the contents of an interface across multiple frames or states. A Prefab *transition* is defined as a pair of *prototypes* and a set of *constraints* that specify when the *transition* is allowable. When an *occurrence* of the first *prototype* is observed, Prefab begins tracking the *transition*. If an *occurrence* of the second *prototype* is observed that is compatible with the specified *constraints*, the *transition* is fired.

3.2 Two Example Models

The next two sections discuss the use and creation of a library of Prefab prototypes, consistently using two example models: a *one-part model* and an *eight-part model*. Prefab is an extensible system designed to support the addition of models that capture new classes of widgets. These two models were therefore selected to provide insight into extension by illustrating a range of complexity in our current models.

The *one-part model* is our simplest model, consisting of only a single feature. A one-part prototype specifies a single exact patch of pixels, and Prefab identifies occurrences whenever it observes those exact pixels. This model might appear to be a strawman, but our experience suggests that it can be quite effective at supporting widgets for which there is not yet a more specialized model. The obvious limitation of the one-part model is that it does not generalize across families of widgets.

The *eight-part model* is illustrated in Figure 3.1 with its parts parameterized by the appearance of the Microsoft Windows Vista Steel button. Features define the corners, a region defines each edge, and constraints require the parts form a rectangle. This prototype's edges are a single repeating pixel, but other region types are possible (e.g., a repeating sequence, a repeating multi-row pattern, a gradient). This

model is designed to describe a wide range of widgets that render a rectangular border (e.g., buttons, containers, menus, toolitps, and entire windows).

3.3 Identifying Interface Elements

Our goal is to identify all occurrences of widgets from our prototype library in an image of an interface. To support real-time interactive enhancements of interfaces, we want to do this many times per second. Because the graphical desktop is not a physical scene, and because we need to identify many widgets many times per second, prior techniques are a poor fit for our problem. For example, Yeh *et al.*'s recently developed Sikuli system uses a combination of template matching and voting based on invariant local features, and it requires a reported 200msec to identify all occurrences of a *single* target [74]. Because we need to more quickly identify all occurrences of *many* widgets, we develop an approach tailored to the recognition of widgets in images of graphical interfaces.

Prefab first conducts a single pass over an image to identify all occurrences of features from the prototype library. Based on the detected features, models generate hypotheses regarding potential occurrences. Actual occurrences are detected by filtering these according to the constraints of the relevant model, including checking the validity of pixels in any of the prototype's region parts. We can also use the identified occurrences to monitor transitions between multiple frames. After identifying occurrences of prototypes in the current image, Prefab determines whether any relevant transitions have occurred and updates its set of transitions that are potentially in progress. For the sake of clarity, this section presents each step in its simplest form, using our one-part and eight-part models as examples. In our later discussion, we note several aspects of the process presented here that can be optimized for performance.

Locating Features

When a prototype library is created, Prefab chooses a hotspot within the patch of pixels defining each feature in the library. Prefab constructs a decision tree for determining whether a pixel in an image of an interface is the hotspot of any feature in the library, as in Figure 3.2. Each internal node specifies an offset relative to the hotspot, each edge corresponds to the color at that offset, and each leaf corresponds to a feature. Traversing the tree to a leaf tests every pixel in a feature (e.g., the leftmost path in Figure 3.2 tests the dark grey pixel, then the blue pixel, then the light grey, and finally the yellow pixel). Some offsets are not relevant to some features, so they will be ignored at the current step of the traversal. If an internal node lacks an edge corresponding to the color at the specified offset, then traversal ends and the pixel to which the tree is currently being applied is not the hotspot of any feature. This decision tree is stored in the library and evaluated against images of interfaces to locate features at runtime.

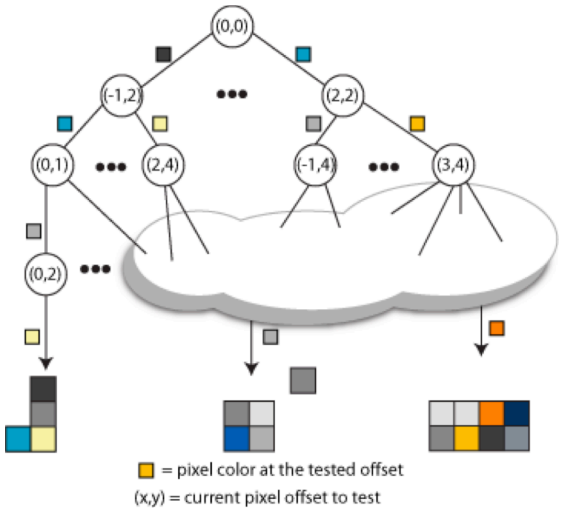


Figure 3.2: Prefab constructs a decision tree that tests whether a pixel is the hotspot of a feature from the prototype library. It uses this tree to scan an image of an interface, detecting all features in a single pass.

The decision tree can be constructed automatically from the library of prototypes in order to optimize performance. Our implementation currently organizes the tree to look for unusual pixels first. For example when choosing a hotspot for each feature in the prototype library, our implementation chooses a pixel of a color that is least common among all features in the prototype library. Specifically, it uses the distribution of colors in the features as a proxy for the distribution of colors in interfaces. These heuristics combine to minimize the tree depth and the length of typical partial traversals.

Generating Hypotheses

After identifying all feature occurrences, each Prefab model generates hypotheses of potential prototype occurrences. Importantly, the overwhelming majority of prototypes in the library have already been removed from consideration. Every model contains at least one feature, and Prefab has identified all occurrences of all features, so any prototype that includes features which have not been detected cannot appear in the current image.

Hypotheses for one-part prototypes are trivial to generate. Because the one-part model consists of a single feature, a single hypothesis for a one-part prototype is generated at each location in the image where that feature occurs.

A naïve but sufficient general method is for a model to enumerate all mappings between a prototype's features and occurrences of those features in an image. For example, a model that contains k features, each of which occurs f times in a particular image, could generate f^k unique hypotheses.

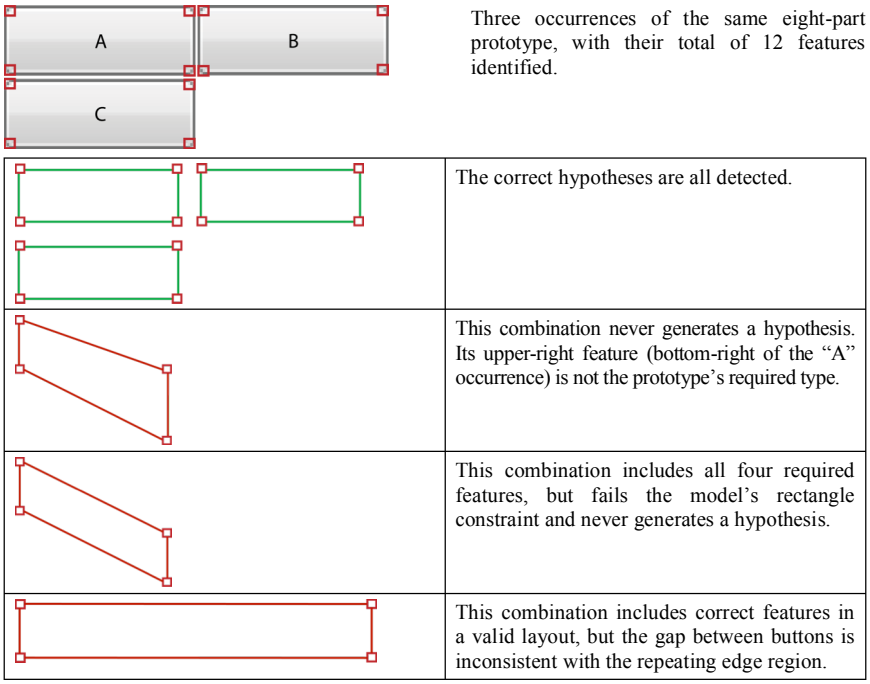


Figure 3.3: Based upon the features detected in an image of an interface, Prefab generates a set of prototype hypotheses and then identifies occurrences by testing those hypotheses against the pixels in the image of the interface.

Prefab models can apply constraints to generate a much smaller set of hypotheses. For our eight-part model, its four features are constrained to a rectangle. The model therefore generates hypotheses by starting from each occurrence of a prototype's top-left feature, checking to the right for the top-right feature, then down for the bottom-right feature, left for the bottom-left feature, and finally confirming the bottom-left feature is directly below the top-left feature. Figure 3.3 illustrates several hypotheses considered by the Windows Steel prototype when it detects features of three adjacent buttons. The number of arrangements considered and the number of hypotheses generated are both far less than the naïve f^k. We have found it relatively easy to implement such methods for efficiently generating small sets of high-quality hypotheses, but more importantly we note that they are implemented in the model and therefore shared by many prototypes. Even if it were relatively difficult to implement an efficient method for generating good hypotheses in a particular model, we believe this effort would be justified if the model could then be applied to a large variety of prototypes.

Detecting Prototype Occurrences

After generating a set of hypotheses, actual occurrences are detected by filtering hypotheses according to the constraints of the relevant model, including checking the validity of pixels in any of the prototype's regions. Hypotheses that pass these filters are reported to applications as prototype occurrences. For example, Figure 5 illustrates six hypotheses considered by the Windows Steel prototype when features were detected in three adjacent buttons.

Prefab's models can use arbitrary code for constraints before or after region validation. In our current models, we have found the combination of constraints on feature arrangement during hypothesis generation and later validation of region pixels to be sufficient and effective. Prefab's support of arbitrary logic for filtering hypotheses may prove useful in future extensions with additional models.

During region validation, models identify sets of pixels to be validated and delegate validation to the region object. In our eight-part model, for example, a hypothesis defines the locations of the four corner features. Based on these, the model defines four sets of pixels corresponding to the top, right, bottom, and left edges. It then delegates validation of each set of pixels to the region objects in the prototype being tested. For the Microsoft Windows Steel button prototype from Figure 3, the region validates the presence of the single repeating pixel along the edge. Different types of regions take their own approaches to validating pixels delegated to them by the model (e.g., a repeating sequence, a repeating multi-row pattern, a gradient). If the region rejects the pixels, the model rejects the current hypothesis.

Monitoring Transitions

Prefab applications can be based entirely upon identification of prototype occurrences, but we have found that many potential applications require observation of a transition from one prototype to another. We therefore added explicit support for observing such transitions with Prefab. Each transition is defined by a prototype whose occurrence initiates monitoring of the potential transition, a prototype whose occurrence indicates it may be appropriate to trigger the transition notification, and a set of constraints. These constraints include both support for arbitrary logic and pre-packaged versions of commonly used constraints, such as timeouts.

Prefab maintains a set of transitions that are potentially in progress. After determining which prototypes occur in the current frame (and reporting them to applications), Prefab checks for transitions that are potentially in progress and could be triggered by an occurrence in the current frame. These are given the option of triggering, subject to their constraints. All transitions in the set are then given the option to expire (separating triggering from expiration allows transitions that trigger and then expire, expire without triggering, or trigger multiple times before expiring). Finally, the current occurrences are examined for prototypes that could initiate a transition and those transitions are added to the set potentially in progress.

3.4 Supporting Prefab Prototype Creation

There are a range of approaches to developing libraries of prototypes required for Prefab applications. A researcher evaluating an interaction technique in a set of existing applications might create a prototype library for the widgets used in those applications. A practitioner or hobbyist who wants to use a Prefab

enhancement with their favorite application might create the necessary library and share it with other people who use that application. Communities might create and maintain large shared libraries, perhaps using wiki functionality like that developed for web mashups with d.Mix [31]. Explicit specification might be enhanced with automated learning of prototypes through passive observation of people using everyday interfaces. We are ultimately interested in all of these possibilities, but this initial work focuses on a core initial requirement of supporting effective creation of individual Prefab prototypes.

It is possible to manually specify the parameters of a model corresponding to a particular widget or family of widgets, but such a process is tedious and likely to introduce errors. We therefore develop methods for the lightweight interactive specification of Prefab prototypes based on positive and negative examples corresponding to widgets that a prototype should or should not identify. The goal is not necessarily complete automation of example-based prototype creation, as we do not believe this is necessary for supporting the effective creation of prototype libraries. Instead, we believe it is sufficient to reduce the effort to a level that it is easy for a person to create a prototype that can be more broadly re-used. For example, a person might click on widget in an image of an interface and then choose from a small sorted list of prototypes (e.g., choosing the prototype that indicates their click was on a slider).

This section discusses lightweight example extraction and our posing of the parameter search problem. Example extraction is described first to illustrate how it is possible to expedite the collection of example images. We then describe how we use branch-and-bound search to recover the optimal prototypes based on these example images [60]. Our search algorithm typically parameterizes a prototype in less than one second. Together these methods can be used to populate a prototype library without any programming or tedious pixel-level manipulations.

Example Extraction

Most example widgets can be quickly extracted from an image of an interface with one or two clicks. Widgets are designed to be easily visible against their background, and so they typically include well-defined edges. A person using our prototype authoring tool captures one or more images of an interface and then provides one or more clicks in the interior of a widget. Allowing multiple clicks accounts for the case where a widget contains multiple apparently disjoint pieces. Prefab then identifies a set of increasingly large rectangles that contain these clicks. Specifically we look for potential boundaries around the click where the gradient exceeds a threshold (the zero threshold defining rectangles where all pixels are the same color). For a particular threshold, all such rectangles can be found in a simple greedy search that is at worst linear in the number of pixels in the image. A simple extension allows finding all unique rectangles for all thresholds in much less than the product of thresholds by pixels. We find all such

rectangles and prioritize the smallest. We then present a sorted list to the person using the authoring tool. For all prototypes we have created, this generates the correct example extraction within the top handful of results. However, we note that tools for arbitrary pixel selection could be provided if a situation were found where these heuristics fail to identify the correct segmentation.

While a person is creating a prototype, our authoring tool applies that prototype to images currently loaded in the tool and displays any detected occurrences. This is helpful for seeing if a prototype is effective, and also eases specification of negative examples. Negative examples are used to correct over-generalization, where the prototype results in false detections. We have rarely encountered situations that require a negative example. One situation arose when recovering a prototype for the custom widgets in the YouTube movie player in Figure 1.1. These are painted such that they share vertical edges (i.e., the pixel defining the right edge of one widget also defines the left edge of the adjacent widget). This led Prefab to over-generalize by interpreting each group of k adjacent occurrences as an occurrence. Marking any one of these as a negative example corrects the prototype by prompting Prefab to base the prototype on the combination of the single-pixel edge and the adjacent interior pixel color.

Model Parameters and Prototype Definitions

Each model exposes a set of parameters, possible values, and constraints on allowable combinations. For example, our eight-part model exposes ordinals for the width and height of each corner feature and for the depth of each edge region.

Importantly, it is *not* a model developer's responsibility to determine optimal parameters (i.e., to optimally divide pixels from an example into parts). A model specifies allowable combinations, and the branch-and-bound search algorithm then chooses the optimal combination. Models have two responsibilities: (1) given a complete parameter assignment, create a prototype and assign it a cost, and (2) given a partial parameter assignment, compute a lower bound on the cost of prototypes that can result. Prefab poses both as a matter of assigning example pixels to model parts.

Creating Prototypes and Assigning Cost

We have found that the most appropriate prototype is typically the one that requires the *fewest pixels* to explain the appearance of positive examples while excluding any negative examples. The intuition behind this is similar to the minimum description length principle, a formalization of Occam's Razor in machine learning [59]. Prefab creates a prototype by minimizing the number of pixels needed to describe each part, then computes the cost of the prototype as the total number of pixels in its parts.

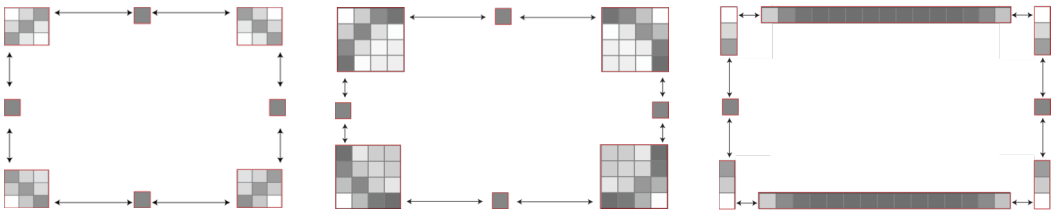


Figure 3.4: These are all valid prototypes for a single example Microsoft Windows Steel button. The left-most prototype is the same shown in Figure 3.1 and requires 40 pixels to define. However the center prototype requires 68 pixels and right-most prototype requires 44 pixels to define. Prefab thus prefers the prototype to the left. Note the left-most prototype generalizes better. For example, the rightmost prototype is restricted to elements of fixed width because its top and bottom edge regions can only identify the exact width described.

For example, the prototype to the left in Figure 3.4 sets the width and height of each corner feature to three and edge depth to one. At these settings, our eight-part model first assigns the nine pixels in each corner to their corresponding features. It then fits a region to the remaining pixels along each edge. The repeating region type can characterize each of the edges with a single pixel, and so is selected by the model. The prototype's total cost is therefore 40 pixels.

In contrast, the center and right of Figure 3.4 show two other prototypes that characterize the same example. The center prototype has larger corner features for which it must pay additional pixels. However, the prototype gains no benefit from modeling corners using these features, because it pays the same number of pixels to explain its edges as the left-most prototype. The right prototype has smaller corners, but the resulting top and bottom edges cannot be described by a single repeating pixel. Because these require 68 and 44 pixels, they are rejected in favor of the left-most prototype. The left-most prototype is also more robust. For example, the right-most prototype above describes only buttons of exactly the same width shown.

Each model takes its own approach to assigning pixels to parts based on a set of assigned values for its parameters. For example, Figure 3.5 illustrates our five-part model of a slider parameterized to identify Mac OS X sliders. The five-part model includes a feature for each end of the trough, a feature for the thumb, and two regions for the trough on either side of the thumb. Based on provided parameters, it first assigns pixels to the end features, then scans the remaining pixels. At each step it either assigns a column of pixels to the left region or skips a fixed number of pixels determined by a thumb width parameter, and assigns them to the right region. Similarly, the eight-part model first assigns pixels to each of the corner features, then scans the remaining edge pixels. For the horizontal top and bottom edges, it iteratively builds a pattern by scanning from the left to the right between the corners. Similarly, vertical edges are built from the top to bottom. Assignments are generally straightforward in both models, as any challenging aspects should be exposed as a parameter and delegated to the search (e.g., the optimal size of edges or the slider thumb).

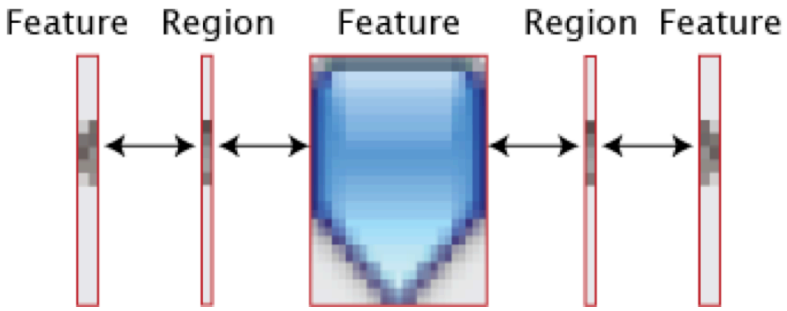


Figure 3.5: This Prefab prototype for Mac OS X sliders is an example of a five-part model. Three features define the ends and the thumb, the two repeating trough patterns are defined as regions, and constraints require the parts to align horizontally. This prototype recognizes all Mac OS X sliders, independent of their width or thumb position.

Multiple positive examples add the requirement that a prototype describe all of them. Specifically, the current assignment starts with one positive example to parameterize a prototype. The model then checks if the prototype matches the other examples and rejects the hypotheses that do not generalize. The model can also look for variations across multiple positive examples to discover pixels that are irrelevant to the prototype and should be ignored. For example, our eight-part model extracts corner features from one positive example and then ignores the pixels that differ from the corresponding corners of the other positive examples. Negative examples override the cost optimization. Specifically a prototype with a lower cost than the current best-known prototype is checked against negative examples before being promoted to the best. There is no reason to check the negative examples otherwise. If a prototype matches a negative example, it effectively has infinite cost.

Computing Lower Bounds on Partial Assignments

Prefab uses a branch-and-bound search, so models must compute lower bounds on partial assignments. For example, if the current best prototype for a particular eight-part model requires 40 pixels and the first several parameters of a partial assignment configure the top corners such that they consume 40 pixels (e.g., making them each 4x5), then no possible assignment of the other parameters can lead to an improvement over the current best prototype.

Models implement lower bound estimates according to how they assign pixels to parts. If enough parameters have been assigned to compute the actual pixels required by one or more parts, the actual cost of those parts can be used. Cost values can also be cached and reused at later steps of the search to avoid expensive re-computation. As an optimization this is possible because the cost of pixels within a particular part is at least as much as any subarea of those pixels. In the models we explored, we found that it is also possible to use previous computations as an admissible heuristic to bound the search. Specifically, if a model has previously evaluated the true cost of a part, that cost can be used as a lower

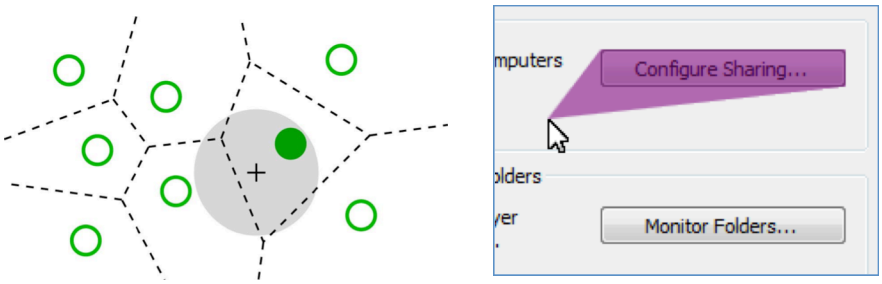


Figure 3.6: Grossman and Balakrishnan's Bubble Cursor expands to ensure that the nearest target is selected [28]. Prefab can enable such target-aware pointing techniques as general enhancements across a variety of applications independent of their underlying toolkit implementation.

bound in any assignment that grows the bounds of that part. Lower bound correctness is required for the branch-and-bound search, so models estimate a single pixel for parts for which they cannot provide a tighter bound.

3.5 Validation in Applications

Prefab demonstrates a new approach to enhancements independent of the toolkits used to implement interfaces. Because this is a fundamentally new capability, there is no reasonable comparison to other approaches for obtaining the same effect. We instead validate, demonstrate, and provide insight into Prefab by implementing and discussing a set of applications. We select these three applications with the goal of illustrating a range of complexity.

All of our applications run on Microsoft Windows and are implemented in Microsoft's C#, using redirection mechanisms as discussed with Figure 1.1. In order to demonstrate Prefab enhancements running on interfaces in Macintosh OS X, we connect via remote desktop software. Prefab thus continues to run on the Microsoft Windows machine, adding its enhancements based entirely on the pixels delivered through the remote desktop connection.

Target-Aware Pointing Techniques

Chapter 2 provides a detailed overview of target-aware pointing techniques and the challenges of implementing them in real-world interfaces. This section presents a rudimentary implementation of the Bubble Cursor to demonstrate the real-time methods presented in this chapter. Chapter 5 then generalizes from these methods to implement a general-purpose Bubble Cursor that runs across an entire desktop. Figure 3.6 shows our implementation in a dialog from Windows Media Player. Figure 1.2 shows the same Bubble Cursor in a Mozilla Firefox dialog on Macintosh OS X and in a YouTube movie player on Microsoft Windows. Our videos at prefab.github.io show a similar implementation of Sticky Icons. To the best of our knowledge, Prefab is the first approach to implementing target-aware pointing

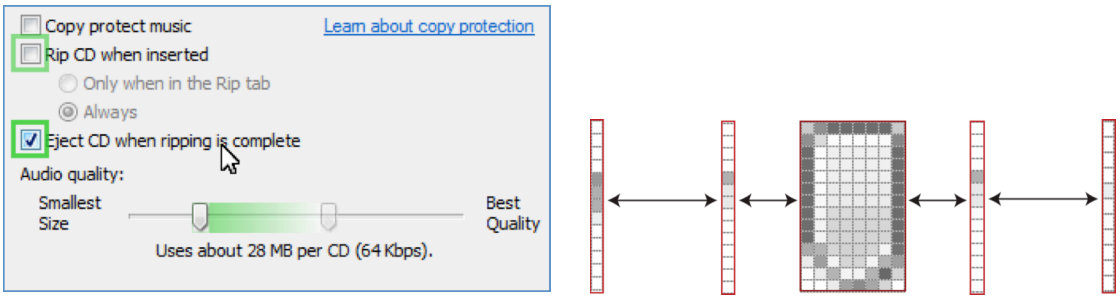


Figure 3.7: We have used Prefab to implement Baudisch *et al.*'s Phosphor [1] based entirely on an interface's pixels. We use the image of the thumb from the slider's prototype to paint the ghosted thumb in the afterglow.

techniques as general enhancements across a variety of existing applications independent of their underlying toolkit implementation.

To implement these techniques, we extracted examples and built prototypes to identify Microsoft Windows and Macintosh OS X buttons, checkboxes, textfields, and other standard widgets. We similarly created several prototypes of more specific widgets, such as the custom slider in the YouTube movie player. These are based on the same models as the standard widgets. These target-aware pointing techniques require only the location and size of current targets, so it was not necessary to implement any transitions. The Sticky Icons enhancement simply adjusts the mouse gain, while the Bubble Cursor annotates the target window and manipulates how click events are mapped to the source.

Phosphor

Baudisch et al. developed Phosphor, which uses afterglows to explain interface transitions [6]. One potential benefit Baudisch et al. identify is in remote collaboration, where the afterglow makes it easier to determine what changes a remote collaborator has made in an interface. Despite the promise of Phosphor, it has been difficult to evaluate its effectiveness in realistic use or to deploy it in collaborative meeting software.

Figure 3.7 shows our Prefab implementation of Phosphor, with afterglow effects showing recent manipulation of two checkboxes and a slider in Windows Media Player on Microsoft Windows. Figure 1.1 shows the same code creating Phosphor effects for Macintosh OS X widgets in Apple's iTunes (using the remote desktop method discussed earlier). Note the afterglow applied to the sliders in these figures includes a ghosted thumb, even though toolkits generally do not expose a method to expose the appearance of the thumb. Also note these afterglows require near instantaneous recognition of widget manipulation. Because Prefab is based entirely on pixels, Phosphor could be included in collaborative meeting software independent of the toolkits used to implement shared applications.

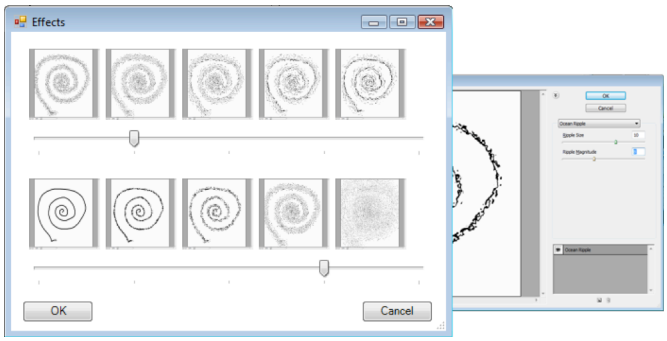


Figure 3.8: We have used Prefab to implemnt Terry and Mynatt's Side Views parameter spectrum previews [67]. Our implementation is a parameter spectrum for an Adobe Photoshop filter running on Microsoft Windows. We populate the spectrum by using Prefab to automatically interpret the interface of Photoshop's filter dialog.

Building upon the same prototypes from our target-aware pointing demonstrations, our implementation of Phosphor uses Prefab transitions to observe widget state changes. These expire and are reset whenever a widget is observed in the same state as the previous frame. A timeout on the transition allows a widget a reasonable time interval to animate into a new state, and we use output redirection to superimpose our afterglow. The ghost of the slider's actual thumb is painted using the thumb feature from the Prefab prototype that detects the slider, as illustrated in Figure 3.7.

Side Views Parameter Spectrums

Our previous applications show local widget enhancement, but Prefab can also be used as part of larger enhancements. To demonstrate this, we implemented Terry and Mynatt's Side Views parameter spectrums [67]. The goal of parameter spectrums is to better support open-ended tasks by simultaneously previewing a range of options for values of multiple parameters. Terry and Mynatt showed parameter spectrums for image editing filters in the GNU Image Manipulation Program (the GIMP), chosen because its open-source implementation allowed necessary modifications. Despite the promise of this technique, the closed-source nature of Adobe's Photoshop prevents researchers and practitioners from studying or deploying parameter spectrums in this more widely-used package. This problem is typical of situations where researchers and practitioners find themselves unable to modify, personalize, or interoperate with widely-used interfaces.

Figure 3.8 shows a parameter spectrum for an image filter in Adobe Photoshop on Microsoft Windows. Our associated video also shows parameter spectrums for the GIMP implemented without modifying its source. In both cases, we use Prefab to interpret the filter dialogs and use synthetic mouse events to manipulate sliders within the filter dialogs to obtain images to populate the spectrums. An eight-part prototype identifies the preview area in each dialog, from which we extract the image for a particular parameter configuration. An interesting difference between the two is how our implementation determines when to capture the preview image for a particular parameter configuration. Photoshop

provides a small progress bar at the bottom-left of the filter dialog, and we use Prefab to observe the completion of this progress bar before grabbing the preview. The GIMP does not provide such a progress bar, and so we monitor the contents of the preview area and capture an image after observing a stable change. In both cases, we use a timeout to recover from the situation where Prefab does not observe the expected transition.

3.6 Discussion and Insights

This chapter represents a first step towards pixel-based reverse engineering of interfaces based on models of how those interfaces are painted by applications and user interface toolkits. This section discusses several important aspects of Prefab and identifies important directions for future work.

This chapter intentionally presents the simplest description of using a prototype library to reverse engineer an image, focusing on information flow as Prefab locates features, generates hypotheses, and tests those hypotheses. There are many opportunities for principled optimizations. The most important is probably that most pixels do not change between successive frames. Prefab's entire process can be implemented with incremental evaluation, using lightweight dataflow to re-compute exactly the features and prototypes that could possibly have changed as a result of differences between successive frames. Prefab's entire process also supports a parallel approach, as feature detection can be applied to multiple pixels simultaneously, multiple models can simultaneously generate hypotheses from detected features, and those hypotheses can be tested in parallel. It is also possible to vastly reduce the number of pixels that must be tested for features in each image. For example, only half the pixels in an image need to be tested if every feature contains at least two adjacent pixels. Given these and other opportunities for principled optimization, we do not foresee performance as problematic for most applications. Our current single-threaded implementation uses a simple ad-hoc frame difference optimization, and our associated video shows multiple applications recognizing prototypes in real interfaces of existing applications with frame-to-frame computations typically well under 50msec.

Not all interfaces are created entirely from standard widgets. Prefab can still be effective in such interfaces for two reasons. First, even non-standard interfaces are procedurally generated. After a prototype of such an interface is created, it is unlikely to change. Second, non-standard interfaces cannot be generally successful unless they *look like an interface*. Prefab may ultimately prove to be a more reliable approach to non-standard interfaces because it is based on their appearance instead of toolkit introspection mechanisms that developers often neglect to implement in non-standard components.

We have shown that a number of interesting applications are enabled by Prefab's core methods, but these methods alone have several important limitations. There is no structured approach to interpreting the content portions of prototype occurrences (e.g., Prefab can identify the three buttons in Figure 3.3, but not the fact that they are labeled "A", "B", and "C"). These methods also do not model relationships among widgets, and so it is difficult to understand the behavior of a group of radio buttons or the hidden content associated with a scroll pane or a tab panel. Finally, it is non-trivial to create sophisticated models. We have stated that this effort can be justified if the resulting model applies to a large and wide variety of prototypes, but developing more complex models can expand Prefab's coverage.

We mentioned that our Phosphor demonstration renders thumb features extracted from our slider prototypes. This highlights an important advantage of pixel-based methods over alternative approaches that expose widget models rather than a widget's on-screen view. For example, the accessibility API intentionally encapsulates the coordinates of a slider's thumb and therefore cannot support Phosphor. Chapter 6 expands upon our Phosphor implementation and presents more general methods for stylizing enhancements based on prototype parts.

We ultimately intend for Prefab to support a wide variety of applications beyond those enabled by these core methods, such as accessibility enhancements, tutorials for off-the-shelf software, context-sensitive help mechanisms, and collaborative extensions to existing applications. Whether Prefab's core methods are suitable depends upon the intended enhancement. In the case of accessibility enhancements, Prefab can enable target-aware techniques with significant implications for people with motor impairments. However, the lack of content interpretation means Prefab cannot be used to implement screen reading for people for visual impairments. The next chapter addresses these challenges with pixel-based methods for recovering content and hierarchy.

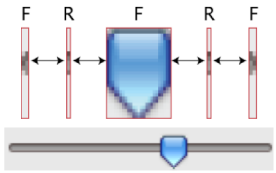
Chapter 4 | **CONTENT AND HIERARCHY**

Prefab's prototype-based methods can be used to identify individual elements within an interface, but they do not describe how to model the *relationships* between elements. This chapter therefore builds upon Prefab's models of widget layout and appearance. Section 4.1 first introduces the use of hierarchy to characterize complex widgets that are composed of multiple elements. Section 4.2 then introduces content regions and shows how they enable efficient recovery of widget content. Section 4.3 shows how these insights can be combined to recover a hierarchical interpretation of an entire interface. Section 4.4 validates these methods in a set of applications that demonstrate capabilities enabled by interpretation of content and hierarchy, and Section 4.5 discusses insights and future work suggested by this work.

Figure 1.2 illustrates several applications enabled by content and hierarchy. Our implementation of Kelleher and Pausch's stencils-based tutorials [44] uses interface hierarchy to robustly reference specific widgets (i.e., differentiating among identical widgets by their position in the hierarchy). Our implementation of Findlater et al.'s ephemeral adaptation [23] leverages our models of content regions to create the necessary gradual onset animations. A translation enhancement highlights real-time identification of interface content by extracting text, translating it, and re-painting it in the interface's original look and feel. Finally, adding customization to an existing interface illustrates opportunities for managing occlusion and rendering new content using our pixel-based models.

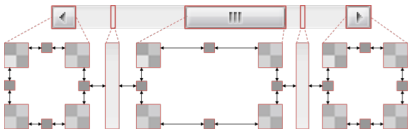
The contributions of this chapter to pixel-based methods are:

- Methods for hierarchical models of widgets. These improve model implementation, example-based prototype creation, and runtime widget detection.
- Methods for modeling widget content regions. These enable efficient runtime recovery and interpretation of widget content.
- Methods for example-based parameterization of widget content regions. This is challenging because examples include content that should not be part of a prototype.
- Methods for recovering the content and hierarchy of an entire interface. This enables DOM-like interpretations of interfaces independent of their implementation.
- Validation of our methods in a set of novel pixel-based applications. These demonstrate our methods and also illustrate opportunities to leverage our methods in addressing other challenges, such as robustly referencing widgets, re-rendering interfaces from pixel-based models, and managing interface occlusion.



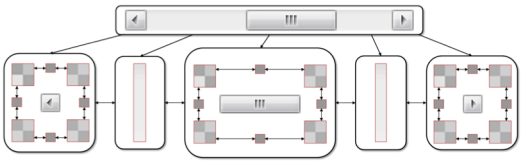
4.1a: Prefab's original methods support a five-part model of sliders.

4.1b: Scrollbars vary their thumb size, so cannot be modeled with five parts.



4.1c: This model accounts for the presence of scroll buttons and the varying size of the scroll thumb, but is too complex for practical use.

4.1d: Other types of scrollbars arrange buttons differently, which would require still greater complexity in the above model.



4.1e: Hierarchical models simplify the implementation, example-based parameterization, and recognition of complex widgets.

Figure 4.1: This chapter introduces hierarchical models of widget layout, improving Prefab's support for complex widgets defined by hierarchies of simpler components.

4.1 Hierarchical Models

The eight-part model in Figure 3.1 was the most complex model explored in the previous chapter. Although Prefab's original methods can characterize many widgets, significant challenges arise when considering more complex models needed to represent widgets consisting of multiple components. Figure 4.1 illustrates this by comparing the modeling of a *slider* with that of a *scrollbar*.

Figure 4.1a shows a five-part model of a slider, consisting of a feature for the thumb, features for the trough endpoints, and regions for the variable-length trough. This model effectively characterizes many sliders. Although a scrollbar might seem to have a similar layout, Figure 4.1b shows that scrollbars vary the size of their thumb to illustrate what portion of the scrollable area is currently within view. A single feature is therefore insufficient for characterizing the scrollbar thumb, so we replace the feature with eight parts describing a thumb of varying size. If we want our model to represent the buttons at either end of the scrollbar, we also need to replace the end features with eight parts. Figure 4.1c shows the resulting model,

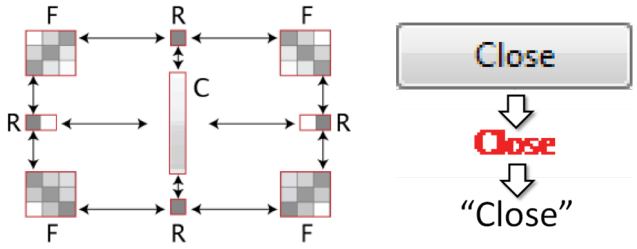


Figure 4.2: This prototype for Microsoft Windows Steel buttons is an example of a nine-part model. My nine-part model is identical to the eight-part model in Figure 3.1 except for the addition of an interior content region. This prototype's content region has been parameterized as a single repeating column. At runtime, content within a button is obtained by differencing the repeating column against the pixels inside the button's content area.

which characterizes the components of these scrollbars but has become too complex for practical use. It is difficult to correctly implement, and its many parameters create a high branching factor that makes it computationally expensive to fit prototypes from examples. The effort to implement and optimize complex models can be justified when they characterize a wide variety of widgets, but this model still does not characterize some common scrollbars. For example, Figure 4.1d shows a scrollbar from Mac OS X's Cocoa toolkit painted with both scroll buttons grouped together to one side of the bar.

We leverage the insight that a hierarchy of simpler widgets typically defines more complex widgets. We introduce hierarchical models of widget layout, as illustrated with a scrollbar model in Figure 4.1e. This extends Prefab's original notion of delegating regions to procedural code (e.g., a gradient or a repeating pattern) by allowing delegation to another model. Portions of a hierarchy can be re-used in implementing multiple models. A model can account for portions of the hierarchy appearing in different arrangements or being optionally absent. The hierarchy can also be used when fitting prototypes from examples and annotations of simpler components constraining a search of the overall hierarchy (e.g., leveraging a label to distinguish the scrollbar thumb). At runtime, hierarchical prototypes are identified by locating simpler components and then testing constraints and regions in the hierarchy.

4.2 Content Regions

Prefab's pixel-based methods are built on the insight that the pixels of a widget are procedurally defined. This is critical to Prefab's real-time performance, as it allows exact feature matching as a basis for prototype detection. But some interface content varies dramatically and cannot easily be identified through exact matching. For example, modern toolkits often employ sub-pixel rendering and anti-aliasing techniques in rendering text. This improves readability, but also modifies text's pixel-level appearance in unpredictable ways. The same characters can be rendered as many different combinations of pixels. Prefab's original methods therefore could not address the recovery of widget content (e.g., Figure 3.1's Windows Steel Button prototype cannot recover the text inside).

We address this challenge by building upon several insights. First, toolkits construct interfaces as trees. Second, content appears at the leaves of a tree (i.e., labels and icons do not contain other widgets). Every piece of content is therefore contained within a parent. Third, the parents of these leaf nodes paint simple backgrounds (often a single color, sometimes a simple gradient). This is critical to interface usability, as a person must be able to easily see the content painted over that background. Instead of directly modeling unpredictable content, we introduce *content regions* that model the much simpler background of a parent and efficiently identify content using runtime differencing.

Figure 4.2 illustrates a nine-part model of a border with an interior content region (i.e., it is identical to the eight-part model from Chapter 3 except for the addition of the interior content region). The model's constraints require that the content region describe every pixel not accounted for by the corner features or the edge regions. In this case, a prototype of the Microsoft Windows 7 Steel Button parameterizes the content region with a single repeating column. At runtime, content is obtained by differencing the repeating column against pixels inside the button's content area. Figure 4.2 illustrates this differencing with red pixels in an example button. A character recognition algorithm is then applied to recover the text "Close". Because such character recognition is relatively expensive, it is important to note that our content region method efficiently identifies a small portion of an interface to which more expensive methods can then be applied. Its computation can also be cached, as there is no need to re-execute an interpretation of identical content pixels.

Parameterizing Content Regions by Example

Recall that Prefab supports the use of *examples* to create prototypes. Parts are assigned by a search minimizing the number of pixels needed to describe those examples in a manner consistent with the model. Like other regions, content regions are modeled as procedural methods for pixel generation (e.g., painting a single color, repeating a pattern, painting a gradient). Prefab's original methods cannot be applied to content regions because each example contains unpredictable content. A simple part cannot characterize this content, and so the search fails to fit a good prototype that generalizes from the example.

We address this problem by defining the cost of a potential prototype as the sum of two components: *model cost* and *content cost*. As before, the model cost is the number of pixels used to define the parts of a prototype. The content cost is the number of pixels in an example that do not match the prototype specified by a content region. The intuition behind this approach is that minimizing the sum requires the search to both *describe* the background and *identify* the foreground. Because we lack a meaningful method for generating that unpredictable foreground, we pay full cost for the pixels it occupies. Note that

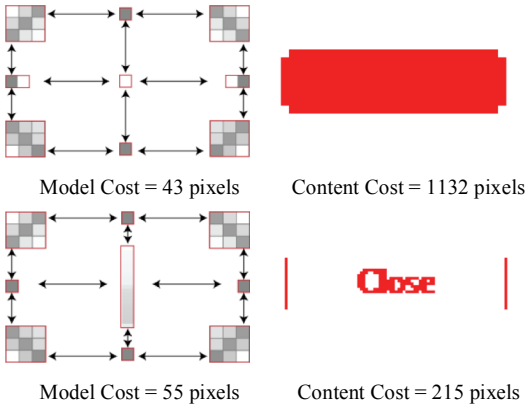


Figure 4.3: These are both valid nine-part prototypes for a single example of a Microsoft Windows Steel button. They incur total costs of 1175 and 270 pixels. Prefab thus prefers the nine-part prototype shown in Figure 4.2, which costs only 246 pixels and is also more general. Note that Figure 4.2's prototype also identifies the correct content.

this is a generalization of Prefab's original method, as content cost is always zero in models without content regions.

As an example, Figure 4.2's prototype sets the width and height of each corner feature to three, top and bottom edge depths to one, and left and right edge depths to two. The content region is a repeating column of pixels. The prototype has a model cost of 57 with these settings (9 for each corner, 6 for the edges, and 15 for the content region). The text results in a content cost of 189 (the red pixels shown on the right side of Figure 5). The total cost is therefore 246 pixels.

In contrast, Figure 4.3 shows two other prototypes the search might consider for the same example. The first has the same corners and edges but attempts to fit a single color to the background of the content region. This improves its model cost to 43, but the poor match results in a content cost of 1142, and a total cost of 1175 pixels. The second example has the correct content region with the corner and edge configuration from Figure 4.3 (which was fit to an eight-part model that does not consider content). Specifically, notice its left and right edges are 1 pixel wide. This results in a model cost of 55 (9 for each corner, 4 for the edges, and 15 for the content region), but the content cost is increased to 215 by two 13-pixel columns at either end of the content region. Its total cost is 270 pixels. These and other prototypes are rejected, with the search ultimately selecting the configuration from Figure 4.2 as the best fit.

Note that the content region in Figure 4.3 has actually resulted in a better characterization of the prototype's other parts. Without a content region, there is no reason for Prefab to determine that the left and right edges of this example are two pixels wide (and so it finds the prototype form Figure 3.1). The inclusion of a content region has in this case led Prefab to produce a prototype that describes *every pixel* in the example. Our validating applications present implications of this more complete interpretation.

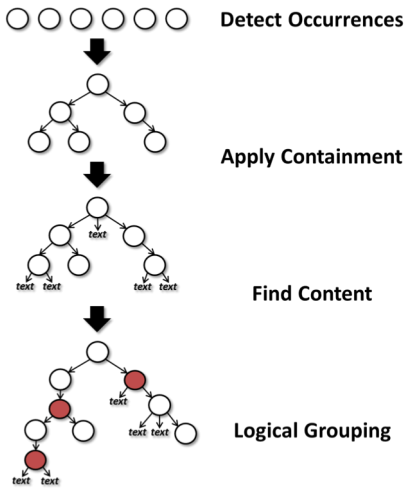


Figure 4.4: We interpret interface content and hierarchy by detecting widget occurrences, applying containment to construct a tree, finding content within widgets, and logically grouping nodes within the tree.

4.3 Interpreting Content and Hierarchy in an Entire Interface

The intuition behind our methods for individual widgets can be extended to support pixel-based interpretation of content and hierarchy in an entire interface. Instead of considering content only in terms of text within a button, the necessary insight is that every widget is content relative to its parent. Our challenge is to recover the content and hierarchy of the entire interface while retaining Prefab's performance. We implement this in four steps, as illustrated in Figure 4.4.

We first apply Prefab's library of prototypes to locate widgets. This uses feature-based detection to identify a set of widget occurrences. We then organize the detected occurrences into a tree. The root is the image itself (typically a top-level window in Prefab's current input and output framework). The tree is constructed using constraints provided by each occurrence's model. These typically enforce spatial containment within a content region of the occurrence. The primary exception is for widgets that float above an interface (e.g., tooltips, popup menus, drop-down boxes). Tagging prototypes of these widgets allows our tree construction algorithm to link them directly to the root.

This organizes occurrences that were detected using our feature-based methods, but we still need to apply our differencing method to locate unpredictable content from content regions. It is important this differencing respect the existing hierarchy (i.e., nested widgets must consider the proper background and must not generate spurious content in areas owned by children). Our current implementation uses a post-order traversal. We generate a composite background image when traversing down the tree, then test and mark pixels when traversing back up the tree. Widgets only test pixels within their content regions that were not accounted for by children. Identified content is interpreted and added as a child of the widget that detected it.

The resulting tree includes all detected widgets arranged by their containment. Additional organization can be added by considering that siblings in this *visual* tree may suggest an additional component in a *logical* tree. For example, several pieces of text might be grouped together and then related to an adjacent checkbox. Prior research has developed methods for semantic grouping of widgets [25]. Given our focus on pixel-based detection of the visual tree, we perform logical grouping using a set of heuristics.

4.4 Validation in Applications

This section presents new methods for real-time pixel-based interpretation of widget content and hierarchy. We validate and provide insight into our work through a set of demonstrations. We select these with the goal of illustrating a range of complexity in applying our methods.

All of our applications are implemented in Microsoft's C# running on Microsoft Windows 7 and using redirection provided by Prefab. We use remote desktop software to demonstrate enhancements running on Mac OS X interfaces. Prefab thus continues to run on the Microsoft Windows machine, adding its enhancements based entirely on the pixels delivered through the remote desktop connection. We apply enhancements to a variety of well-known applications to highlight that our methods are independent of the underlying implementation.

Stencils-Based Tutorials

Kelleher and Pausch's *stencils-based tutorials* provide help directly within applications using translucent stencils with holes to direct a person's attention to the correct interface component [16]. Such an enhancement is difficult to broadly deploy because of the rigidity and fragmentation of existing applications and toolkits. It is beyond the capabilities of previous pixel-based systems because authoring such a tutorial requires support for referencing *specific* interface elements. For example, there may be several buttons of identical appearance within an interface, but only one of them is the appropriate next action.

Figure 1.1 and our videos on prefab.github.io show our Prefab implementation of stencils-based tutorials. The tutorial instructs a person on how to download résumé design templates in Microsoft Word 2010. The video for content and hierarchy highlights the real-time responsiveness enabled by our new methods.

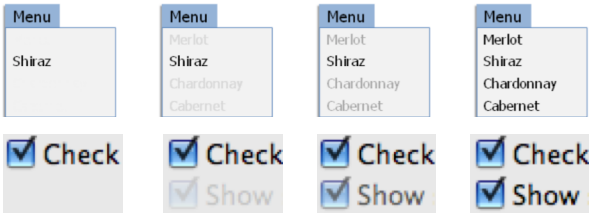


Figure 4.5: Findlater *et al.*'s ephemeral adaptation technique uses the gradual onset of unlikely targets to facilitate easier targetting of likely targets [23]. This image shows Findlater *et al.*'s original implementation of a menu testbed together with our pixel-based implementation within a Skype dialog running on Mac OS X.

Stencils-based tutorials are a straightforward application of widget hierarchy. Knowledge of the full hierarchy allows us to reference widgets using simple path descriptors on the tree. We implemented this demonstration by building prototypes to identify the majority of widgets in Microsoft Office 2010. For example, we used nine-part models to characterize many of the containers and buttons. We also used one-part models to identify less structured content (e.g., the icons used to represent different types of templates). These one-part models are typically easy to construct (i.e., a model of the background of their parent makes it trivial to segment the one-part example). We converted Prefab's hierarchical interpretations into an XML format, allowing the use of XPath descriptors to reference widgets within the hierarchy. Tutorials are thus authored as a list of XPath descriptors paired with textual instructions for each step. Additional capabilities could be developed, and we have not yet explored the best approach to an authoring tool, but this demonstration highlights our use of the pixel-based hierarchy to reference specific widgets.

Ephemeral Adaptation

Findlater *et al.* developed ephemeral adaptation, an adaptive technique that improves performance by reducing visual search time and maintaining spatial consistency [5]. Ephemeral adaptation helps draw visual attention to likely targets in an interface. Specifically, likely targets appear as normal within an interface, but unlikely targets are initially missing and then slowly fade in. Despite the promise of this technique, it has been difficult to evaluate in realistic use or to widely deploy in everyday software. A pixel-based implementation is beyond prior systems for two reasons. As before, it requires the ability to reference specific widgets (e.g., to monitor how frequently they are clicked in order to model which are likely targets). In addition, this application requires the ability to remove unlikely targets from the interface and then render their gradual onset.

Figure 4.5 and our prefab.github.io videos show our implementation of ephemeral adaptation using Prefab within a Skype settings dialog box running in Mac OS X. Upon moving between tabs, likely targets in each tab are initially visible. Unlikely targets then fade in over time. This enhancement uses a nine-part prototype of the tab pane and various prototypes for each of the interior widgets. We use our

XPath descriptors to tally the frequency of interaction with each widget and identify the most commonly-used widgets in each tab.

We render the gradual onset animation using the content region from the tab's nine-part model. Specifically, we render tab background (i.e., the pixel-level appearance of its content region) as an overlay at the location of each unlikely widget. We then gradually fade this overlay from opaque to transparent. This creates the illusion that the widget is gradually fading into view. Note this technique requires identifying all of the content throughout the interface in order to appropriately animate its onset, a capability not supported by prior pixel-based methods.

Language Translation

In addition to pixel-based *identification* of interface content, our methods can help enable real-time *interpretation* of interface content. To demonstrate this, we implemented a pixel-based enhancement that automatically translates the language of an interface and then presents the translated content in the same look and feel as the original interface. Because of the rigidity and fragmentation of current tools, interfaces usually must be translated by their original developer (or somebody else with the application source). Our methods allow anybody to translate an interface and could thus form a basis for community-driven translation (similar to advances in social accessibility [65]). To the best of our knowledge, ours is the first method for real-time translation of interfaces independent of their underlying implementations. Although translation is not the same as complete localization, it is an important step.

Figure 1 and our associated video show our translation enhancement applied to a Google Chrome Options dialog running on Microsoft Windows 7. The left image illustrates a portion of the original dialog in English. The right image shows that same portion of the dialog with the text translated into French. Our associated video on prefab.github.io also includes a Spanish translation of the same dialog. We implemented this by interpreting textual content identified by our methods, translating that text, and then rendering the new text in the original interface. Our associated video shows this enhancement running in real-time. This requires identification and interpretation of content occur quickly enough to handle the appearance and movement of content within a scroll pane.

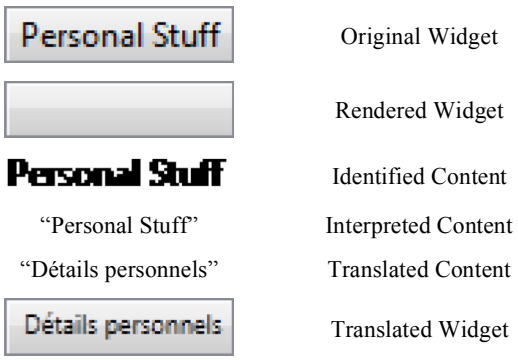


Figure 4.6: We use Prefab's methods for interface content and hierarchy to implement a translation enhancement that preserves the look and feel of the original application. A translated widget is rendered by compositing the translated text with its prototype's content region.

There are several potential approaches to interpreting screen-rendered text [30]. My current implementation uses an ad hoc template matching method, leaving integration of more advanced methods as an opportunity for future work. Importantly, our methods separate the identification of text from interpretation of that text. The interpretation of a region of pixels can thus be cached to eliminate potentially expensive re-interpretation of those same pixels (e.g., using a hash of the pixels). In our video, each piece of text is interpreted only the first time it appears. We then translate it using a machine translation service. As the text moves within the scroll pane, our content detection recovers the same pixels and retrieves the text from cache.

To maintain the application look and feel, we paint the translation into the original interface. This is implemented by using each widget's content region to render an overlay masking its content (i.e., its English text). The translated text is then rendered within the bounds of the original content region. For example, Figure 4.6 shows a button before its translated text is rendered. Because English text is typically shorter than translated text, we adjust the font size of the translated text to fit in the available region. Our next demonstration explores a more sophisticated modification of the interface to accommodate new content.

Interface Customization

Our pixel-based interpretation of interface hierarchy also provides a framework for modeling some common forms of *occlusion* in interfaces. Occlusion is at the very core of the desktop metaphor, as it allows interfaces to limit the complexity of presented interfaces via the illusion that additional portions of the interface continue to exist even when they are not visible. For example, tab controls use occlusion to limit attention to related subsets of complex interfaces. Existing pixel-based methods are strictly limited to interpreting visible portions of an interface. The need to observe an interface is inherent to pixel-based

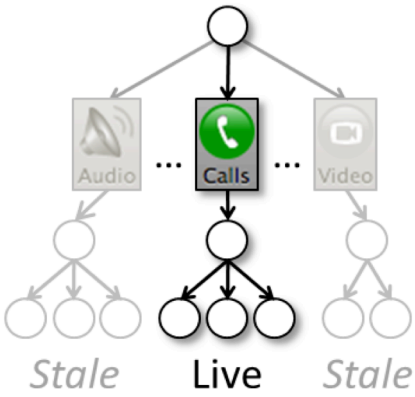


Figure 4.7: We use our knowledge of hierarchy to manage widget occlusion in a tab pane. We store the most recently observed image of each widget, annotating occluded widgets to indicate those images are currently stale.

methods, but we can use our knowledge of interface hierarchy to help manage common forms of occlusion.

Figure 1.1 and our associated video at prefab.github.io present a demonstration of this in the context of interface customization. Instead of automatically adapting an interface according to widgets that are likely to be used, this example allows people to manually flag widgets as "favorites" for quick access. Figure 1 shows this applied to the same Skype dialog box from our Ephemeral Adaptation example. A small star is added to each widget in the interface. Clicking this star adds the widget to a "favorites" tab we added to the interface. Viewing that tab presents all starred widgets and allows interaction with each of them. As with all of our demonstrations, this is implemented using pixel-based interpretation with input and output redirection.

The management of occlusion is inherent to this example. My implementation enhances the hierarchy to store the most recently observed version of each tab (using our XPath descriptors to reference each tab and its contents). We annotate these nodes in the hierarchy as *stale* to capture the fact they are occluded. Figure 10 depicts a simplified snapshot of the interpreted hierarchy with occluded nodes. If a "favorite" widget is currently occluded in the source window, it is painted using its stale version from the hierarchy. When a person moves to interact with a widget, synthetic input events are generated to bring that tab of the source window into view (i.e., to ensure that portion of the hierarchy is responsive to interaction via standard redirection mechanisms). Synthetic events could also be generated to regularly poll stale portions of the hierarchy, but this was not needed in our demonstration.

This example also demonstrates the use of our pixel-based methods to add new elements to the interface. Nine-part models of the window, the tab button container, and the tab pane are used to create a larger version of the window, insert the new "favorites" tab button, and to render the blank tab area into which

“favorite” widgets are added. Figure 1 and our associated video show the added tab button and the extended window. We view this as an initial peek into opportunities to go beyond overlays and more fundamentally transform the rigidity and fragmentation of existing interfaces.

4.5 Discussion and Insights

This chapter advances state-of-the-art pixel-based systems by presenting the first methods for real-time interpretation of interface content and hierarchy. We now briefly discuss some important aspects of our pixel-based interpretation and identify some insights gained from this exploration that we address in the following chapter.

Chapter 3 presented the simplest description of our pixel-based methods for modeling widgets via prototypes. This chapter expands upon those methods for interpreting interface content and hierarchy. A variety of optimizations could improve performance. For example, the entire interpretation process can be implemented using lightweight incremental evaluation to compute exactly the sub-tree of the hierarchy that could possibly have changed between successive frames [11]. Given this and other potential optimizations, we generally do not expect performance to be problematic in most applications. Our implementation currently uses parallelization when interpreting content. We currently re-compute the entire hierarchy whenever Prefab identifies new features. Our associated video shows multiple demonstrations that interpret content and hierarchy in real interfaces of existing applications with computations between frames typically under 100msec. We believe this is sufficient for the applications we explore.

The preparation of our demonstrations highlighted another advantage of our approach to interpreting interface content and hierarchy. If our methods are applied to an interface that contains widgets that are not already in Prefab’s prototype library, the parents of those currently unknown widgets identify the pixels of the unknown widgets as content. We used this fact to quickly extract the examples used to create prototypes for our demonstrations, and we believe it could provide a basis for an improved prototype authoring tool.

Our translation demonstration currently uses an ad hoc approach to text interpretation (exact matching against a library of labeled character snippets). Off-the-shelf OCR technologies are generally ineffective because of the extreme low resolution of typical interface text. We previously noted the availability of recognition methods for screen-rendered text [30], but these are not optimized for Prefab’s scenario. A deeper investigation of robust text interpretation methods is an opportunity for future work.

We have noted the existence of prior work examining logical grouping of interface elements [7]. It is unclear whether these methods are compatible with the real-time requirements of pixel-based interpretation. The tree-based organization of interfaces and our ability to interpret visual containment provides an important advantage: we can likely consider only logical groupings of siblings. Based on the hierarchies we have encountered in our work, there are typically a small number of siblings in any given node (i.e., most interface elements are intentionally designed to contain a small number of content items). Our current implementation uses simple heuristics to perform logical grouping. For example, checkboxes are matched to their corresponding content using a threshold on the proximity of the nearest text. Future work can explore more advanced approaches to creating logical groupings by matching elements of an interface. Errors in an automated process could be also corrected by storing annotations that record the need for a specialized grouping (e.g., using our XPath descriptors to override the default behavior).

This chapter focuses on core methods for interpreting content and hierarchy together with demonstrations of their value in example applications. There is a significant opportunity for future work that more thoroughly characterizes these and other pixel-based methods. Such work might examine the variety of widgets encountered in applications, how well pixel-based methods can characterize those widgets, how many types of models and parts are necessary, and which of those models and parts are most broadly effective. Our pixel-based methods are the first to rival the accessibility API in terms of completeness, so comparisons between our methods and the accessibility API may be appropriate. Such a comparison should preferably go beyond simple frequency of failure to also probe the nature of failure (e.g., its impact in applications, the difficulty of correcting a failure). As in Hurst *et al.* [36], there are likely significant opportunities for hybrid approaches that combine the strengths of the accessibility API with the strengths of pixel-based methods. The contributions of this chapter are a necessary step toward future characterizations of pixel-based methods, and our current validations are appropriate for this work.

Our interface customization demonstration dynamically re-renders a dialog box at a different size to create room for the “favorites” tab. This is possible because the nine-part prototype that detects the dialog box describes all of the pixels needed to generate it. To the best of our knowledge, we are the first to demonstrate pixel-based methods to create new widgets matching an existing interface. The ability to seamlessly add new widgets to the interfaces of existing applications would dramatically extend pixel-based methods, and Chapter 6 provides more structured support for stylizing widgets based on our prototype-based representations.

Our customization demonstration illustrates one approach to managing occlusion (using the hierarchy to maintain a memory of occluded components). Tab controls are perhaps the simplest case and this method

may not immediately generalize to other forms of occlusion. For example, popup menus create less predictable occlusions that can span multiple nodes in a hierarchy (because they float above the hierarchy). Occlusion within a scrollpane also presents different challenges. We have shown that interface content and hierarchy provide a useful framework for reasoning about occlusion, and future work could examine more advanced methods building upon these initial insights.

Chapter 5 | **TARGET-AWARE POINTING**

Chapter 3 and Chapter 4 validate Prefab with some initial demonstrations of pixel-based enhancements. However, there are many challenges in developing enhancements that work outside of the isolated interfaces explored in those chapters. In this chapter and the next, we surface and address these challenges by implementing two interaction techniques that function across the *entire desktop*. We also report on the limitations of these *general-purpose* enhancements unearthed by examining them in the complexity of real-world interfaces. We therefore contribute important progress toward real-world deployment of interaction techniques like these and shed light on the gap between understanding techniques in controlled settings versus behavior with real-world interfaces.

5.1 A Real-World Bubble Cursor

The human-computer interaction literature includes many promising techniques for *target-aware pointing*, including deep studies of specific characteristics of those techniques in controlled laboratory settings. This important family of techniques can improve pointing for a variety of people on a range of devices in many applications. Target-aware techniques can significantly outperform other pointing facilitation techniques, and they ultimately have great potential to improve the efficiency of interaction.

Despite the promise of these techniques, few have been deployed or evaluated in real-world interfaces. The impact of target-aware pointing, and our understanding of its effectiveness, is currently limited by two challenges to *implementing* target-aware pointing in real-world interfaces.

First, external pointing enhancements often cannot obtain reliable information about the size and location of interface elements [14,36]. As mentioned in Chapter 2, accessibility APIs attempt to provide some of the necessary information, but are inevitably incomplete due to developer failures to implement the API. The API also exposes widget *models*, not necessarily their on-screen view (e.g., the pixel coordinates of a slider's thumb are intentionally encapsulated). Techniques for dynamic code modification can also often be made to work in a single application or toolkit [20,21], but are generally too brittle for enhancement of the full desktop. Code injection techniques also sometimes fail (e.g., when a skinnable application uses pre-rendered images of widgets instead of meaningful graphics operations). Failures at these levels of applications can only be corrected by their developers, so many interface elements remain opaque.

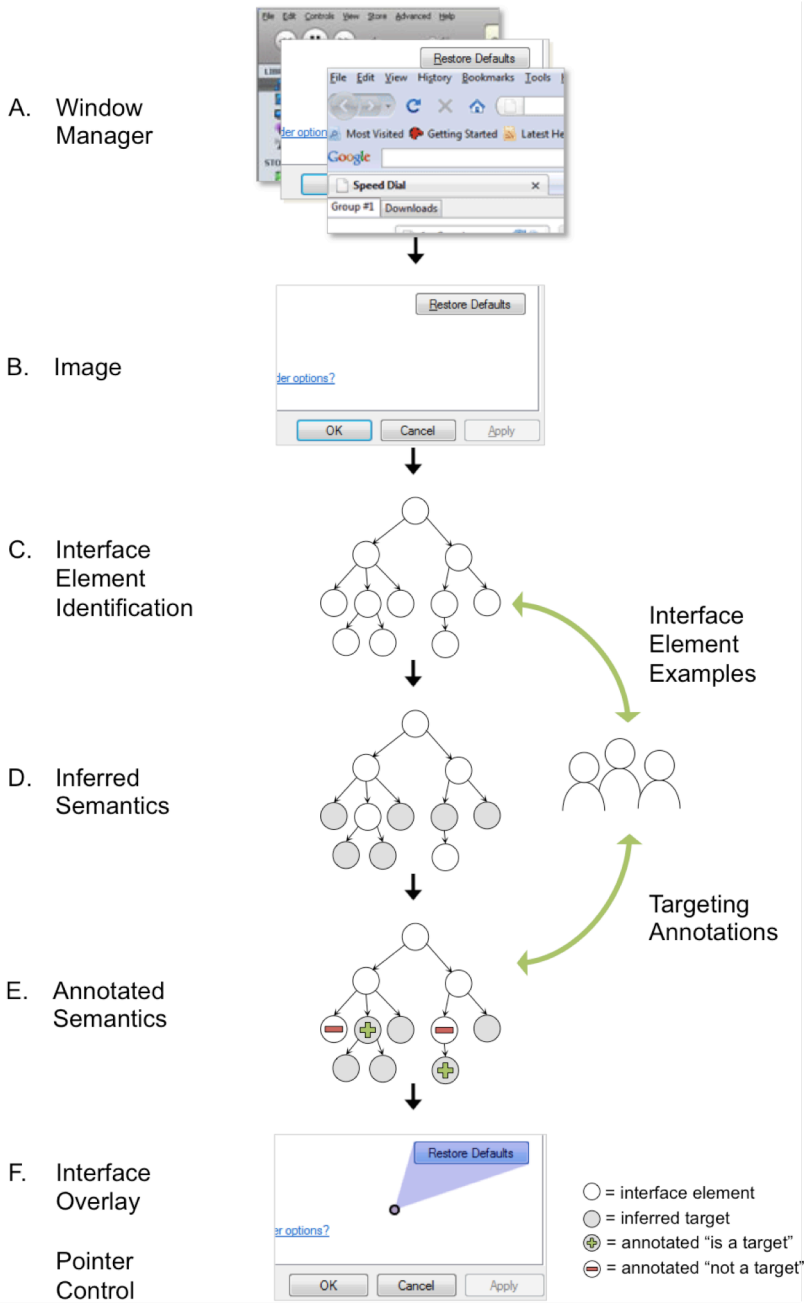


Figure 5.1: We implement a general-purpose Bubble Cursor in a novel architecture that combines pixel-based *identification* of interface elements with *interpretation* of their targeting. We emphasize a *human-driven* approach with interactive extension and correction of both implementation layers.

Second, even a complete enumeration of interface elements is insufficient for determining how a targeting enhancement should behave. Studies of pointing facilitation techniques generally treat an interface as a field of abstract targets (e.g., gray circles), but the notion of a target is often not well defined in real-world interfaces. Targeting ambiguities are presented by calendars, paint canvases, text fields, and many other standard and custom widgets [72]. It is difficult or impossible for the developer of a general-purpose enhancement to foresee and address all such ambiguities.

We address these challenges in a general-purpose implementation of Grossman and Balakrishnan's Bubble Cursor [28], generalizing our demonstration from Chapter 3. Our implementation functions across the Windows 7 desktop, as illustrated in Figure 1.1. The methods we develop can be applied to any modern desktop, and they can be extended to support other target-aware techniques. Specifically, we architect our enhancement to separate *identification* of interface elements from *interpretation* of how to target those elements. We then implement identification using Prefab's methods for reverse engineering interface structure. We implement interpretation by annotating interfaces with desired targeting behavior. We also develop interfaces for correcting errors in both levels, and we argue that *social* mechanisms for identification and interpretation are essential to any broad target-aware deployment.

Figure 5.1 offers an overview of our system. It first queries the window manager for images of targeted windows. It then *identifies* elements using Prefab's pixel-based methods. This requires a library of Prefab prototypes, each created from one or more example images. Prefab creates a tree containing a node for each interface element (e.g., a leaf for a text label, a leaf for a slider thumb, an inner node with children for a button and any interior icon or text). It then *interprets* the interface to determine potential targets, using annotations inferred from the interface structure and content together with annotations from prior interactive target correction. Finally, it walks the tree to determine what the Bubble Cursor should target and then overlays a translucent highlight over that target. Executing this process many times per second yields the general-purpose Bubble Cursor.

Among the variety of target-aware pointing techniques, we implement the Bubble Cursor for several reasons: (1) we believe it is the fastest general technique in the literature, (2) it can be implemented as an overlay, without modifying targeted elements, and (3) it is exemplary of the family of target-aware techniques, so the methods we develop should have broader relevance. Other target-aware techniques include gravity wells [39], force fields [1], sticky icons [73], semantic pointing [8], area cursors [42], enhanced area cursors [22], bubble targets [15], object pointing [30], and drag-and-pop or drag-and-pick [5]. Such techniques offer great potential, but remain difficult to deploy in practice. This chapter's discussion section revisits this, emphasizing this dissertation as a starting point for the real-world design and deployment of target-aware techniques.

My methods are inherently *human-driven*, in that we focus on allowing people to improve targeting by interactively correcting erroneous behavior. We also expect any broad deployment will be *social*, with people sharing interactive corrections and receiving updates based on the corrections of others. Interfaces are procedurally generated, so their pixel-level appearance rarely changes. Familiar interfaces will therefore be thoroughly annotated and appear to "just work". But interactive correction will remain

important, both to ensure individuals can correct private, niche, or unpopular interfaces and to allow communities to quickly annotate newly-released interfaces.

The specific contributions of this chapter include:

- An implementation of a general-purpose target-aware pointing enhancement. Specifically, we scale Prefab's implementation of the Bubble Cursor using pixel-level analysis to target interface elements throughout Microsoft Windows.

- A novel architecture for general-purpose target-aware pointing enhancements. Informed by the insight that knowledge of interface element locations and dimensions is insufficient for a general-purpose target-aware pointing enhancement, we separate pixel-level *identification* of interface elements from higher-level *interpretation* of how to target those elements.

- Two interfaces for end-users to correct the behavior of a general-purpose target-aware pointing enhancement. We examine the requirements and design space for such interfaces and develop two concrete designs that explore complementary approaches.

- A discussion of implications for the design and evaluation of target-aware pointing techniques in the complexity of real-world interfaces. Examining our implementation with real-world interfaces reveals both unexpected targeting behaviors not addressed in prior literature and new potential approaches to overcoming the limitations of current target-aware techniques.

Interpreting an Interface

Chapter 3 and Chapter 4 described how Prefab identifies a hierarchy of elements, but any hierarchy is by itself insufficient for targeting. Even complete compliance with a typical accessibility API does not enable an effective Bubble Cursor. This is ultimately due to mismatches between available metadata and the needs of an external enhancement. Framework and application developers cannot foresee all potential external enhancements, so they cannot provide all relevant metadata. For example, current accessibility APIs are designed to provide access to the underlying data in each widget (i.e., the widget *model*). They therefore expose the value to which a slider is set, but not the screen location of the thumb (precluding targeting of that thumb). General targeting behavior is undefined for other elements (e.g., calendars, paint canvases, text fields), and will vary among techniques. We believe this mismatch between available metadata and the needs of enhancements is inherent, requiring a human-driven approach.

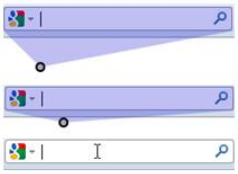
My system therefore allows people to interactively *annotate* which elements of an interface hierarchy should be targeted by a Bubble Cursor. Any node can be marked as a *target*, can implicitly not be targeted due to other targets in the hierarchy, or can be explicitly marked *not a target*. We store annotations using an XPath-like path descriptor based on properties of an element, its location in the hierarchy, and properties of its ancestors. A library of annotations therefore consists of path descriptors with associated metadata. Annotations can be quickly retrieved for an element, and entire libraries can be stored, combined, and shared to enable social annotation.

A path descriptor needs to identify a *specific* element in a *specific* interface while being robust to changes in element size or position. The root of a hierarchy corresponds to the entire image processed by Prefab, so the root needs to include how an image was captured. We currently set root attributes for the application executable name and top-level window class. If this became insufficient, additional root attributes could be added for problematic applications (e.g., including the URL for images captured within web browsers). The remainder of the descriptor is based upon the unique identifier for each prototype along the path from the root to the annotated element. Elements identified via background differencing do not have a prototype, so we use a content attribute based on a hash of their pixels (which allows differentiation among sibling content elements). Finally, we use an index for otherwise identical descriptors.

Direct manual annotation likely can be sufficient in a broad deployment leveraging social mechanisms, but also can be expedited by even minimal inference. We currently employ two simple mechanisms. The first assumes every leaf is a target, a heuristic that yields good behavior for many widgets (e.g., checkboxes, radio buttons). The second generalizes annotations of inner nodes to other nodes with identical subtrees. Scrollbars provide one example of this, as their thumbs vary in size and usually contain a knurling graphic. A nine-part model represents the thumb as a node with a single child (i.e., the knurling). Preferred behavior is to target the entire thumb, and annotation of one scrollbar can then generalize over other occurrences of that scrollbar. Arbitrarily sophisticated inference could identify likely targets, and our XPath-like descriptors suggest relevance of the deep literature on wrapper induction [45]. But any inference mechanism will sometimes fail, so we see these techniques as a powerful complement to human annotation.

My implementation chooses a target for the Bubble Cursor in a pre-order traversal of the hierarchy of the window nearest the pointer. We consider each *target* node to determine which is closest to the pointer, defining distance to be zero if the pointer is within a target. Recursion ends at target nodes, because the spatial containment represented by the hierarchy means any children will be further from the pointer.

Recursion can therefore also end at non-target nodes that are further than the current best. Some arrangements require considering the possibility the nearest target is not in the nearest window, but other windows are typically ignored because their root is further from the pointer than the current best target.



My design for real-world interfaces also required subtle refinement of the Bubble Cursor. One important refinement is our degradation into the standard point cursor whenever within a target (illustrated here with a storyboard of the cursor approaching a textbox). On one hand, this is aesthetically simpler than crosshairs used to illustrate the center of the original Bubble Cursor with abstract targets [28]. But it is also important because it creates a *graceful degradation* in the face of unknown interface elements or ambiguous targets. If Prefab fails to interpret the pixels in a portion of an interface, or if the appropriate targeting behavior is genuinely ambiguous (e.g., as with a text field), then normal point cursor behavior is automatically restored in that region without requiring a hotkey or other modifier. Pointing can therefore be improved in many interactions without necessarily being penalized in others. We believe such *conservative* strategies are promising for the design of deployable external interface enhancements.

The cursor must also be capable of *dragging* targets while moving or sizing elements (e.g., scrollbars, sliders, windows). Dragging is implicitly supported by our conservative strategy, as moving into a target allows use of the point cursor to drag the target. We also enable dragging from outside a target. When the drag is initiated, the Bubble Cursor latches onto the target. Subsequent movement drags the target, and mouse release resumes standard targeting behavior. We currently do not use knowledge of potential drop targets during the drag, but could imagine strategies for target-aware drops.

Implementation Details

This section has focused on novel methods and strategies in our system, as these can be applied and extended in future research on target-aware pointing or external enhancement of existing interfaces. For completeness, we briefly discuss relevant details of our current Windows 7 implementation.

At the start of each cycle, we query the Windows 7 Desktop Window Manager for the bounding box, application executable path, class name, and z-order for each visible top-level window. We construct a hierarchy with the desktop at the root and top-level windows as children, and we then capture pixels for the window closest to the pointer. If two windows are at the same distance (typically due to overlap), we choose the front. As a performance optimization, we ignore entire windows that are marked as non-targetable (e.g., visible but non-interactive windows). We also infer target annotations during traversal, and only when there is no existing human annotation (i.e., our inferred annotations are lazily evaluated).

We render our Bubble Cursor as a translucent window, and update its z-order to render above the targeted window. We redirect input using a low-level hook to intercept mouse events and update the center of our Bubble Cursor instead of the system pointer. We use each mouse event to move the center of our Bubble Cursor, and we clip the system cursor to the center of our targeted element. If the target is partially occluded by another window, we clip to the center of a visible region (and adjust the size of our overlay).

Our associated video on prefab.github.io was captured on a typical laptop. Prefab is mostly single-threaded and unoptimized, but still processes an interface every 100 msec. Many of the potential performance optimizations described in our previous chapters could improve performance, however our video shows the cursor is responsive and we have not found performance to be a concern.

5.2 Interactive Targeting Correction

We previously noted that many familiar interfaces will seem to "just work", but that interactive correction remains important to our approach. We identified six requirements for correcting target identification and interpretation:

- *Invoking Correction:* A person must be able to access an interface to correct erroneous targeting behavior.

- *Capturing an Interface:* Errors are caused by incorrect identification or interpretation of elements in images, so a person must be able to capture images that illustrate the failure condition.

- *Visualizing Identification and Interpretation:* A person must be able to understand what error occurred so that they can determine how it should be corrected.

- *Extracting Examples:* A person must be able to extract examples of the elements of an interface that should be identified by our pixel-based methods.

- *Creating Prototypes:* A person must be able to use extracted examples to create appropriate prototypes that can then be identified at runtime.

- *Authoring Annotations:* A person must be able to annotate desired targets.

These requirements imply a large design space. Because we focus on the first deep implementation of a general-purpose target-aware technique, we explore two initial points in the design space. The first is a full-featured design, called the *Target Editor*. The second is a lightweight interface for in-context annotation, called the *Annotation Menu.*

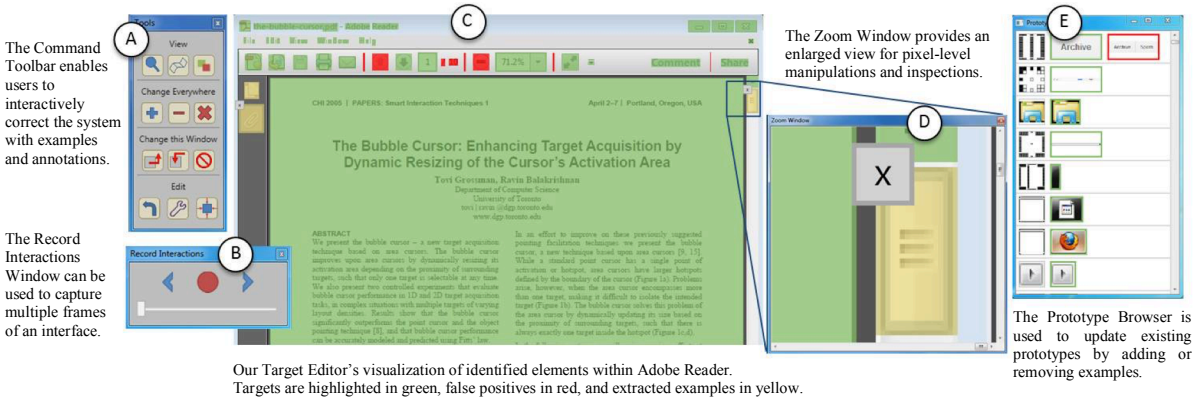


Figure 5.2: The Target Editor is used to extract examples, create and update prototypes, and manipulate annotations.

Correcting Behavior with the Target Editor

Figure 5.2 presents screenshots of the Target Editor. Its major components include the *window image*, colored *highlights* of identified elements, and the *command toolbar*.

Invoking Correction. The editor is always accessible via a system tray icon and a keyboard shortcut. When a person encounters erroneous targeting, they invoke the editor and then click a window on the desktop to "edit" that window.

Capturing an Interface. Selecting a window as part of invoking the editor triggers capture of an image of that window. The image is processed to identify and interpret interface elements, then displayed in the editor.

Some errors only occur during interaction with a *dynamic* interface, so a captured static image may be insufficient. For example, the blinking text cursor in a textbox can lead to erroneous targeting. When present, it is identified as a leaf and targeted. When absent, the textbox itself becomes a leaf and is targeted. The result is a Bubble Cursor that jumps between the blinking caret and the larger textbox. To determine why this occurs, a person may need to inspect multiple images captured at different times.

The editor addresses this need with a "Record Interactions" tool to capture videos of interaction. A record button starts and stops recording, a slider supports skimming captured video, and buttons advance individual frames. The blinking text cursor behavior is fixed by annotating the textbox as a target, thereby overriding the inferred targeting of the text cursor leaf.

Visualizing Identification and Interpretation. Translucent highlights visualize system identification and interpretation of interface elements. Green highlights are placed over elements that will be targeted (i.e., the first *target* node encountered on every path from the root, whether inferred or annotated). Red highlights are placed over elements explicitly annotated as *not a target*. Identification is thus encoded via

the presence of a highlight, while interpretation is illustrated via color. Interactive selection is illustrated via a border, and elements can be manipulated individually or in batch using multiple selections with the toolbar.

Extracting Examples. Prefab often builds a good prototype from just a single example of an interface element. But it can also under-generalize, requiring additional positive examples to broaden its concept, or over-generalize, requiring negative examples to narrow its concept.

For example, this YouTube movie control causes Prefab to overgeneralize 

because its widgets are painted to share vertical edges. A single example creates a prototype that over-generalizes by identifying every group of k adjacent widgets (creating duplicate leaves and inner nodes that do not describe any true element). Providing any of these false detections as a negative example will correct the prototype, resulting in a prototype that detects the combination of the single-pixel edge and the adjacent interior pixel.

Examples can be specified by rubber-banding a yellow highlight. Each highlight can be moved, resized, or deleted. Because pixel-level selection can be tedious and error-prone, the editor provides "Zoom" and "Snap" tools.

"Zoom" opens an enlarged view of the image and highlights. The enlarged view receives input, so it supports pixel-level adjustment of highlights. As a shortcut, double-clicking in the image or on any highlight also opens the "Zoom" tool with its view centered at the double-click point.

"Snap" resizes highlights to tightly fit interface elements. Informed by a heuristic from Chapter 3, it starts from the center of a highlight and uses gradient thresholds to search for possible bounding rectangles. Among the heuristically-identified rectangles, it chooses the one closest in size to the drawn highlight. This does not always snap exactly to interface elements, but is helpful for expediting extraction. When it fails, it usually snaps close to the desired size and can then be adjusted using "Zoom".

Two categories of examples can be automatically extracted: (1) positive examples of elements already identified by background differencing within a parent's content region, and (2) any negative examples. In both cases, Prefab has already identified the bounds of the element. There is no need to manipulate the highlight, so a person instead just adds the extracted element to a new or existing prototype.

Creating Prototypes. A new prototype is created using the "Add" tool. A dropdown menu allows model selection (e.g., the exact-match, five-part, and nine-part models discussed previously). For each selected highlight, the editor creates a new prototype from the highlighted positive example. Automated

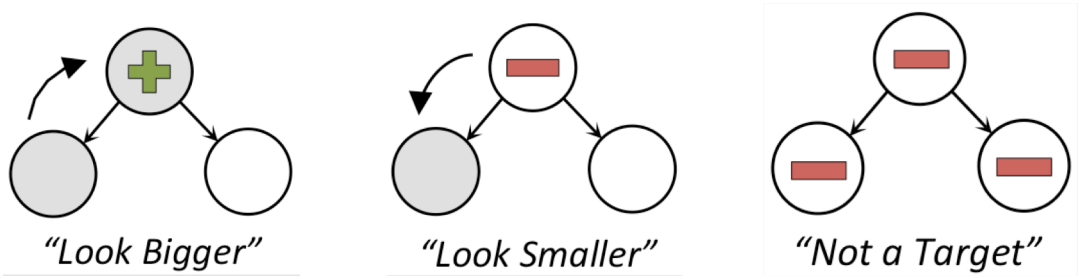


Figure 5.3: Three tools are used together for annotation of desired interpretation of targets. The "Look Bigger" tool annotates the parent of a selected element as a *target*. The "Look Smaller" annotates an element as *not a target,* which causes Prefab to target a descendant. The "Not a Target" tool annotates an element and all of its elements as *not a target.*

creation of the prototype from the example takes a few seconds, after which the user can immediately see the impact of the new prototype on identification and interpretation within the interface.

Existing prototypes are updated with additional negative examples using the "Mark Incorrect" tool. Note that the editor already knows which prototype should be updated: the prototype that falsely identified the example.

Updating existing prototypes with additional positive examples requires the selection of the prototype to receive the new example. The "Prototype Browser" visualizes the prototypes in the current library, together with the positive and negative example images used to create each prototype. A person selects an existing prototype, then uses "Add" to provide it with additional examples.

Bad examples or entire prototypes can be deleted via the "Prototype Browser". Bad prototypes can also be deleted directly in the editor by selecting a highlight and clicking the "Remove" tool. This removes the prototype that identified the highlight, and it is helpful for quickly deleting prototypes accidentally created from bad examples.

Authoring Annotations. Three tools are used together for annotation of the desired interpretation of targets, as illustrated in Figure 5.3. The "Look Bigger" tool annotates the *parent* of a selected element as a *target*. The Bubble Cursor then targets that parent, thus no longer targeting the element. For example, this can be used to target a button instead of its text, a canvas instead of its content, or an entire window instead of any interior icons. The tool does not modify any annotations of the selected element or its children. Such annotations have no effect, as our Bubble Cursor targets downward from the root. Leaving them intact allows "undo" via "Look Smaller".

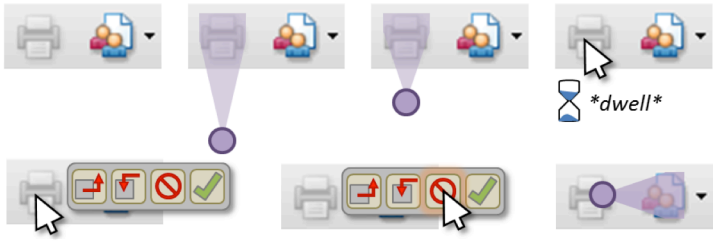


Figure 5.4: A storyboard of the Annotation Menu. It appears when the center of the Bubble Cursor dwells on a target.

The "Look Smaller" tool annotates a selected element as *not a target*, which will cause the Bubble Cursor to consider its descendants as possible targets. Because we heuristically infer leaf elements to be targets, this is typically used to undo the "Look Bigger" tool.

The "Not a Target" tool annotates a selected element and all descendants as *not a target*. This is used to ignore an entire subtree of non-targetable elements, such as labels or disabled buttons.

Lightweight Targeting with the Annotation Menu

The Annotation Menu examines a design that is less capable than the Target Editor, but has the benefit that it can also be used without leaving the context of an interaction. Figure 5.4 storyboards the menu, which is embedded in the Bubble Cursor and *invoked* after a dwell of the pointer within a target.

Interface *capture* and *visualization* are implicit, as the Bubble Cursor itself conveys the current identification and interpretation. Editing is limited to *authoring annotations* using the same "Look Bigger", "Look Smaller", and "Not a Target" functionality from the editor. Each tool makes its edits to the annotations, closes the menu, and updates the Bubble Cursor to target the new nearest target.

Discussion of Interface Design

Figure 5.5 recaps our two designs in the context of our six requirements. They take different approaches to correction, but both are helpful and suggest additional possibilities. For example, we use simple visualizations of the identification and interpretation of elements, but other tree visualizations could be considered. Our *confirmed* annotation was developed for the Annotation Menu, but it would be a sensible addition to the Target Editor. Rapid batch confirmation of targeting would then prevent the Annotation Menu from appearing in interfaces that are already known to be correct. Greater collection of explicit confirmations would also provide additional data for use in training learning systems to automatically infer targets.

Requirement	Target Editor	Annotation Menu
Invoking	System Tray Keyboard Shortcut	Dwell
Capturing	Static Frame on Launch Record Interactions Tool	Implicit
Visualizing	Colored Highlights of Elements	Implicit
Extracting	Rubber-Band Zoom Snap Automatic via Prefab	N/A
Creating	Add / Remove / Mark Incorrect Model Selection Prototype Browser	N/A
Authoring	Look Bigger Look Smaller Not a Target	Look Bigger Look Smaller Not a Target Confirm

Figure 5.5: The target editor and the annotation menu highlight two initial points in the design space of correctional interfaces for the system.

Our requirements and designs also suggest opportunities for fundamentally different approaches. Image *capture* might be automated through passive observation of interface use (e.g., as in Hurst *et al.*'s prior work [36]), with an interface *invoked* later to review targeting corrections inferred from the observed usage. Instead of focusing on the current interface, a design might focus on *creating prototypes*. Such a design could be organized around review, creation, and modification of prototypes and could retrieve images of specific interfaces from a large usage history to illustrate the targeting that results from the current prototype library. A crowdsourcing interface might focus on local *invocation* and *capture*, with other requirements addressed by remote workers. Because our designs require an understanding of pixel-based methods, other designs might explore a range of required *expertise*. Our initial focus has demonstrated effective interfaces, but a variety of additional approaches can be developed upon our underlying technology.

5.3 Examining-Behavior in Real-World Interfaces

The Bubble Cursor and other target-aware techniques are often designed, discussed, and evaluated using fields of abstract targets and distractors. Because our implementation provides a unique ability to deploy such techniques, we sought to examine what new insights we could gain from examining the Bubble Cursor in real-world interfaces. We identified and annotated elements in a variety of interfaces, including Microsoft Office applications, Mozilla Firefox, Adobe Reader, instant message clients, several web pages, and the file browser. We studied a total of 31 applications, creating a library of 754 prototypes and

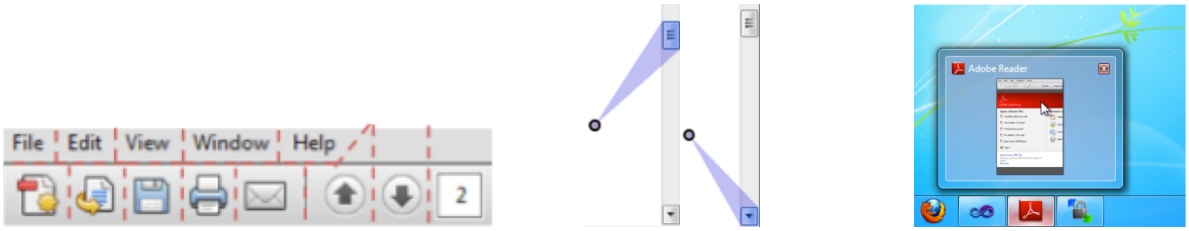


Figure 5.6: We identified limitations in the Bubble Cursor's assumptions around pointer proximity and user intent. These three examples illustrate scenarios where the assumptions fail. The left shows an instance where a person may expect similar elements to have similar targeting behavior in a row of elements. The middle and right examples illustrate confusing behaviors that arise due to dynamics of real-world interfaces.

714 annotations. Our findings with real-world interfaces include insights into two important challenges: (1) the limitations of pointer proximity as a proxy for intent within an interface, and (2) conflicts between target-aware behavior and the intentional design of interfaces for typical point cursors.

Pointer Proximity and User Intent

The Bubble Cursor and other target-aware techniques use pointer proximity as a proxy for intent to manipulate interface elements. It is understood to be an imperfect proxy (hence the notion of distractors), but examining behavior in real-world interfaces gives new insight into limitations.

One insight is that a person may expect similar elements to have similar targeting behavior. Rows of elements is a good example, as the arrangement suggests each item is equally relevant. But placement of a menu bar next to a toolbar can create different effective targets for otherwise similar elements. The Voronoi overlay to the left in Figure 5.6 shows "Page Up" is more difficult to target than "Page Down", and "Help" behaves differently than the other menu items. The behavior is correct, but can be jarring.

The dynamics of real-world interfaces also expose gaps between pointer location and intent. Consider using a Bubble Cursor to click a scrollbar arrow, as shown in Figure 5.6. A person's intended focus is likely to remain upon the arrow after a click, but the movement of the scrollbar toward the cursor may now mean the thumb is the nearest target. Latching onto the thumb is the correct behavior for the Bubble Cursor, but it may be unexpected.

A third case emerges when interfaces act upon a notion of intent that competes with the Bubble Cursor. For example, the Windows 7 taskbar contains a row of buttons for accessing windows of open applications. A hover over any opens a preview, as shown to the right in Figure 5.6. Approaching the taskbar with the Bubble Cursor results in the cursor snapping to a button, invoking the preview, and then snapping to the preview. This is helpful if the desired window is captured, as clicking will activate the window. But if an adjacent taskbar button was desired in the first place, then the benefits of the Bubble Cursor are destroyed by snapping to the preview.

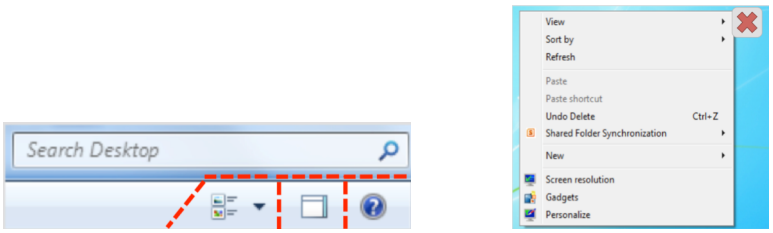


Figure 5.7: Here are two scenarios that highlight conflicts due to the fact that interfaces were intentionally designed for a standard point cursor. The left illustrates how the Bubble Cursor can greatly distort the importance of minor elements, such as those below the commonly used search box in the Windows file manager. The right shows how context menus would require an additional dismiss icon, because the cursor can only select explicit targets.

Conflicts Due to Intentional Design for a Point Cursor

Modern interfaces are carefully and intentionally designed for use with a typical point cursor. This section presents examples where the Bubble Cursor conflicts with that design, together with initial thoughts on how such conflicts could be resolved through extensions to our system.

Designers often use small elements for minor or infrequently-accessed functionality. The Bubble Cursor can greatly distort the importance of such minor elements. For example, the Windows file browser shown to the left in Figure 5.7 includes several minor icons beneath its search box. These eclipse targeting of the search box itself when approaching from below, which is the typical direction of approach. Selection of the search box is therefore more difficult than intended. Extending the methods in this work, it is interesting to consider a *look past* annotation. This might indicate an element should be targeted only if the pointer is directly overhead. The Bubble Cursor could then look past these elements to allow easy targeting of the search box, but would still latch onto minor elements when the pointer was directly overhead. This seems preferable to a hotkey or some other method for disengaging the Bubble Cursor.

Another example is the design of typical context menus, which are dismissed by clicking in non-interactive space. Such a design is incompatible with a technique like the Bubble Cursor that always targets the nearest interactive element. This problem motivated creation of DynaSpot [13], which adjusts the maximum reach of a Bubble Cursor based on pointer speed. Examining this in the context of our system suggests additional strategies. For example, a *dismissable* annotation might overlay a dismiss icon at the upper-right of an element. Upon selection, the system could use its identification and interpretation of the surrounding interface to click in non-interactive space.

A third example is the scrollbar. As pictured in Figure 5.6, it is easy to imagine the utility of a Bubble Cursor attached to a scrollbar thumb. But the trough is also interactive, meaning the thumb can only be grabbed from a narrow horizontal channel. In our own use, we therefore annotate the trough as *not a target*. This allows easy capture of the thumb, but also prevents targeting the trough. It is interesting to realize that the same *look past* annotation discussed above would remove the need for this tradeoff

(i.e., latching onto the thumb from outside, but targeting the trough from within). This also suggests a more general distinction between "major" versus "minor" targets, a perspective that can inform the design of future techniques. Such future techniques could be evaluated with a combination of laboratory studies and insights gained from examination in real interfaces using the methods we have developed here.

5.4 Discussion

Although we describe our approach to implementing a Bubble Cursor, we also provide a general strategy for implementing interface enhancements. Our separation of identification from interpretation, together with our approaches to each, can enable and inform a variety of future enhancements. For example, Prefab prototypes interactively created with our Bubble Cursor can be shared and re-used with any Prefab-based application. Similarly, our Bubble Cursor can benefit from Prefab prototypes created in other applications. At the interpretation level, target-aware enhancements may be able to directly share and re-use annotations (e.g., the notion of a *target* for Sticky Icons [73] may be the same as a *target* with our Bubble Cursor). Other techniques may have different interpretations of targets based on additional annotations, perhaps bootstrapped through inference based upon annotations collected with our Bubble Cursor or some other technique. Finally, our strategies also apply to non-pointing enhancements. For example, Findlater *et al.*'s [23] Ephemeral adaptation could be implemented as an overlay using analyses of interaction history together with explicit annotation of element groupings and other preferences.

Our experiences validate Prefab's methods, but also suggest opportunities to advance pixel-based systems. For example, we found that Prefab would benefit from additional sophisticated models of the pixel-level appearance of complex elements (e.g., it could not deeply model complex text panes found in Microsoft Visual Studio 2010). We also sometimes found it tedious to create multiple prototypes for different appearances of the same interface element (e.g., multiple states of a button, checkbox, or radio button). Pixel-based methods could therefore benefit from additional modeling of the dynamics of interface elements. This chapter has also not addressed the case where changes to a prototype might orphan existing annotations (i.e., might change the descriptor needed to retrieve existing annotations). Chapter 7 explicitly provides structured support for capturing robust annotations, and also explicitly migrating annotations between hierarchies obtained with different prototype libraries (e.g., by checking that they refer to the same spatial region in the captured image). Overall, we believe that deep applications like that pursued in this chapter have a unique potential to inform future development of underlying pixel-based methods. Building out this application has also given us a better understanding of effective and sustainable abstractions for developing with Prefab.

Chapter 6 | **STATES AND STYLES**

Our implementation of the Bubble Cursor is one the most sophisticated applications of any pixel-based methods to date. However, even the Bubble Cursor is no more than a simple overlay that primarily points at and highlights existing elements. At least two key challenges have limited prior enhancements: (1) modeling a widget's *behavior*, and (2) capturing an interface's *style*. Structured support for these two tasks would enable more advanced modifications to a variety of existing interfaces.

Many potential enhancements require an understanding of how an interface changes across multiple frames (e.g., if a checkbox becomes checked, if a slider thumb has moved). In addition to monitoring changes, more advanced enhancements also require manipulating the interface (e.g., sending click events to set a checkbox, dragging a slider). As a result, developers are faced with implementing their own complex frame-to-frame analysis of interfaces together with low-level input redirection mechanisms.

Capturing an interface's style is important because many runtime modifications do not fully alter the appearance of an existing interface, but instead directly overlay new elements onto the interface. Therefore it is important that these enhancements are styled to preserve consistency with the existing interface. Without proper support for capturing style, enhancements are likely to be jarring and unusable.

This chapter addresses these limitations with novel methods for modeling the state and style of individual widgets. Specifically, it builds upon our methods for interpreting interfaces presented in previous chapters. We first introduce methods for real-time modeling of widgets in multiple states. It then describes methods for linking an original widget's state into the representation needed by a new surface enhancement. Finally, it introduces style mappers and show how they can be used to render entirely new widgets in an existing interface while maintaining a style that is consistent with the original design. We focus on these widget-level methods because they serve as building blocks for complex enhancements to entire interfaces.

We contextualize these contributions in an implementation of Moscovich et al.'s Sliding Widgets, touch widgets activated by sliding a moveable element [47]. My implementation dynamically overlays Sliding Widgets on mouse-based interface elements throughout Microsoft Windows 8. For example, Figure 1 and our associated video present screenshots of our implementation in a variety of popular interfaces, including Microsoft Word, Adobe Reader, Gmail in the Google Chrome Browser, and Windows Explorer. We replace elements in these applications using various types of Sliding Widgets, including

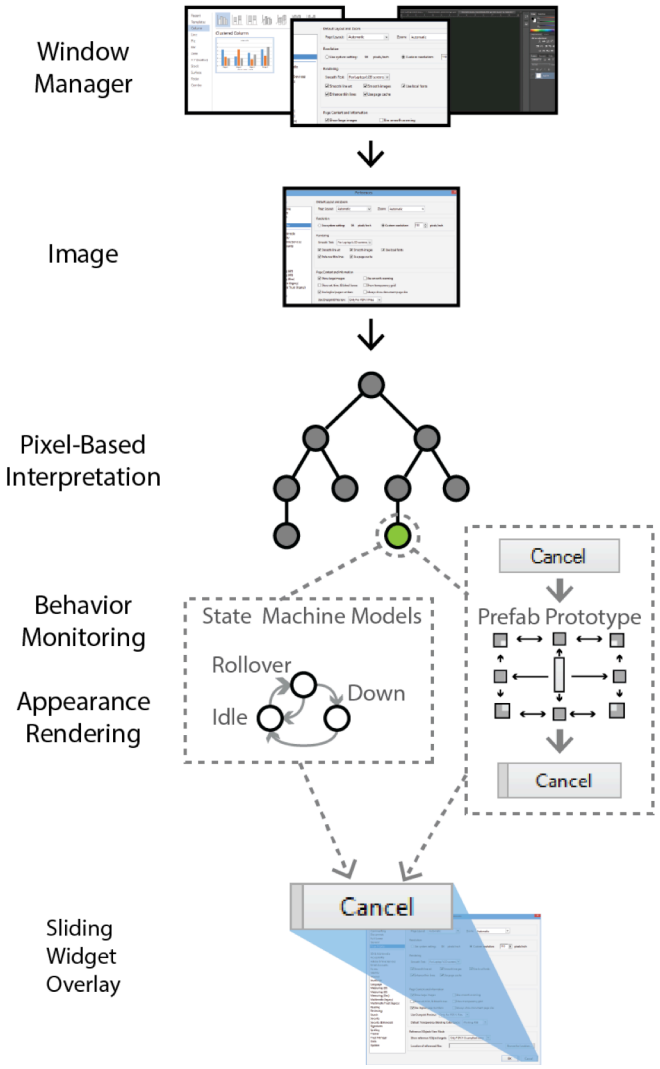


Figure 6.1: This chapter builds upon the previous methods for pixel-based identification of interface elements. Identified elements are replaced by overlaying a corresponding Sliding Widget. The original widget is then modeled and manipulated using state machine models. Finally, we style the appearance of overlaid Sliding Widgets by mapping elements of pixel-level appearance from Prefab's prototypes of the original interface.

Sliding Buttons, Sliding Spinners, Sliding Toggles, and Sliding Dropdown Menus. Sliding Widgets are well beyond the capabilities of prior pixel-based systems, and the implementation in this work informs and validates the design of our new methods.

In addition to validating these methods, we choose to implement Sliding Widgets in the context of existing interfaces because the implementation could directly benefit an emerging set of *hybrid* devices that support both touch and mouse input. Examples of these devices are shown in our associated video. Unfortunately, graphical interfaces for hybrid devices are either difficult to use or expensive to produce. This is because developers either: (1) provide standard mouse-based interfaces with small, dense targets, or (2) implement an entire alternate interface optimized for touch. We therefore explore the approach of dynamically replacing existing mouse-based elements with touch controls. This approach can improve

interaction with hybrid devices without the burden of designing and implementing two entire alternate interfaces.

Figure 6.1 overviews the system. Prefab first queries the window manager for images of windows, and then interprets their elements using the methods described in Chapter 3 through Chapter 5. Specifically, we use Prefab to recover an interface hierarchy containing a node for each interface element (e.g., a leaf for a text label, an inner node with children for a button and any interior content). We walk this tree to determine what elements to replace, and then we overlay our Sliding Widget. To maintain a style consistent with the source interface, our overlay renders each Sliding Widget using pixels extracted by Prefab's prototypes. This process modifies a single frame captured from an interface, and so we repeat this many times per second, synchronizing each Sliding Widget with its underlying widget. Specifically, we use state models to monitor and manipulate underlying widgets. Transitions in a state model cause the Sliding Widget to update its appearance and behavior, and interaction with a Sliding Widget generates input manipulating the underlying widget.

The specific contributions of our work include:

- Methods for pixel-based modeling of widgets in multiple states. Specifically, this chapter introduces *abstract state models* for describing how to interpret and manipulate categories of widgets. Abstract models are then parameterized with pixel-level data specifying the appearance of a specific widget in each of the modeled states.

- Methods for managing the combinatorial complexity that arises in creating a multitude of runtime enhancements that can apply to a multitude of widgets. Specifically, we show how runtime enhancement can be framed in the Model-View-Controller pattern, and we introduce *linkers* for specifying how a particular runtime enhancement relates to a particular abstract state model.

- Methods for styling runtime enhancements to preserve consistency with the design of an existing interface. Specifically, we introduce *mappers* that use pixel-level elements of an interface's appearance to style runtime enhancements to match that interface.

- An implementation of Sliding Widgets as a runtime pixel-based enhancement that can be applied to existing interfaces. This validates our new pixel-based methods and also unearths implications for the design of Sliding Widgets and future pixel-based runtime enhancements.

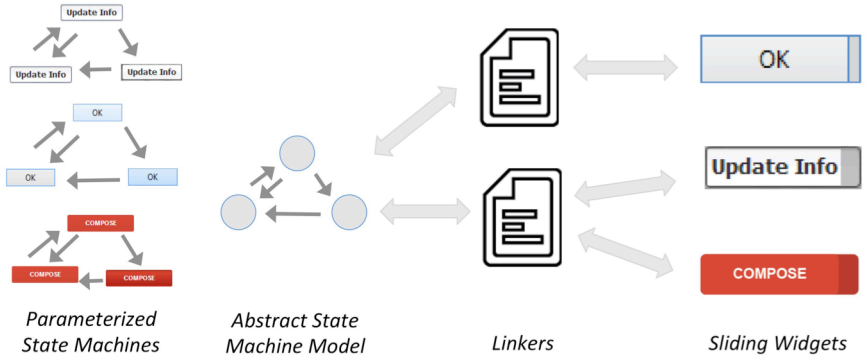
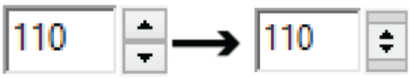


Figure 6.2: Abstract State Models and Linkers decouple the logic of translating widgets to Sliding Widgets, thus avoiding the combinatorial explosion of direct translation

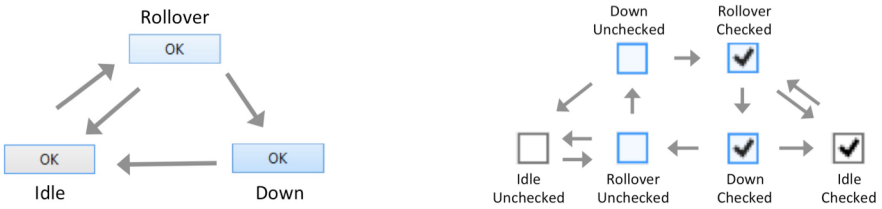
6.1 Interpreting Replacement and Sliding Widget Orientation

Chapters 3 and 4 describe how interface elements are identified from raw pixels. Chapter 5 expands on this using path descriptors to annotate elements with targeting metadata. In this case, Sliding Widgets require metadata indicating (1) which elements should be replaced and (2) the direction in which a replacement Sliding Widget should slide. This chapter takes an approach similar to Chapter 4 to obtain this metadata. Specifically we combine automated interpretation of Prefab's recovered hierarchy with social annotations, wherein people interactively correct erroneous behavior. Interfaces are procedurally generated, so their pixel-level appearance rarely changes. Familiar interfaces will therefore be thoroughly annotated and "just work". But Prefab's combination of automatic interpretation and social annotations allows end-users or designers to interactively correct newly-released or niche interfaces.

For manual annotation, any node can be tagged with a type of Sliding Widget to replace it (e.g., *sliding button*, *sliding dropdown menu*). Nodes can be explicitly tagged *do not replace*, or can be implicitly unavailable for replacement due to replacement of an ancestor. For sliding direction, elements can be tagged *left*, *right*, *up*, *down*, or a diagonal direction (e.g., *down right*). Finally, we allow *horizontal* and *vertical* tags for multi-function Sliding Widgets such as this spinner replacement. For each annotation we store a path descriptor based on properties of an element, its location in the hierarchy, and its ancestors. At runtime we retrieve annotations by matching against each node's path. Thus, we first tag the set of nodes annotated with replacement metadata, and then specify the direction for those nodes.



Social annotation can be sufficient in a broad deployment that leverages social mechanisms, but can be expedited by even minimal automation. We currently include a layer that automatically tags nodes with a sliding direction. This layer iterates over elements marked for replacement and assigns directions in a rotating clockwise manner. We ignore elements in dense layouts, and leave widgets in sparse layouts with a default *left* or *horizontal* direction. Finally, the layer ignores any element with a direction assigned by



Button State Machine
Windows 8 Default Button

Checkbox State Machine
Windows 8 Default Checkbox

Figure 6.3: Abstract State Models are parameterized with prototypes describing the appearance of a widget. These two state models are parameterized with prototypes generated from Windows 8 Default Buttons and Checkboxes.

the preceding exact-match layer, thus allowing end-users or designers to manually override the default. The later section on Examining Sliding Widgets presents examples of interfaces where it is desirable to override sliding direction due to unexpected usability issues in real-world interfaces.

6.2 Modeling and Linking Widget States

Because Prefab captures and interprets individual screenshots, defining the behavior of Sliding Widgets requires monitoring an interface many frames per second and then programmatically redirecting input to manipulate that interface. For example, when a person activates a Sliding Widget by moving its thumb, the system must send a click to the underlying original element. And if that button or any other widget then becomes disabled in response to the interaction, the Sliding Widgets should mirror this change in their behavior and appearance.

The solution to these challenges comes from the insight that a change in appearance of an interface element corresponds to an underlying change in widget state. We therefore model widget dynamics using finite state machines. Although a widget may not be explicitly implemented with an internal state machine, the event-driven nature of interfaces means such a state machine is generally implicit (i.e., widgets exist in some state and change that state in response to interaction events). State machines offer an explicit model of the dynamics of interface elements, which we use to provide *getters* and *setters* for monitoring and manipulating widgets. Specifically, getters allow enhancements to subscribe to events fired when there is a traversal across one or more edges (e.g., signifying rollovers, clicks, drags). Setters can prepackage logic that sends input to an element, transitioning it to a new state.

The critical challenge in modeling state machines is that there are both: (1) a wide variety of existing widgets to be translated into Sliding Widgets, and (2) several types of Sliding Widgets that each require a different mapping. A naïve approach would therefore create a combinatorial explosion translating each widget to each Sliding Widget. We instead decouple parts of the translation using an approach similar to a Model-View-Controller pattern. A *state model* is the model, a Sliding Widget or other enhancement the

view, and a *linker* the controller that synchronizes state models of the original interface with the representations used in a pixel-based enhancement. Figure 6.2 illustrates this decomposition.

Abstract State Models

An abstract state model is defined as a combination of states and transitions describing the behavior of a class of widgets. This state model is then parameterized with prototypes describing the appearance of a particular widget in each of the states. For example, Figure 6.3 presents two state models parameterized to represent behavior for the Windows 8 default button and checkbox. The same models could be parameterized with different prototypes to represent different buttons or checkboxes.

Abstract state models build on Prefab's initial support for observing a transition from one prototype to another [17]. Each transition is defined by a prototype that initiates the transition, a prototype that triggers the transition, and a set of constraints. At runtime, Prefab checks for transitions that are in progress and could be triggered subject to their constraints. After given the option to trigger, these transitions are then given the option to expire (i.e., removed from the set in progress). Finally, the current frame is examined for prototypes that initiate new transitions, which are added to the set in progress.

An abstract state model composes multiple transitions, each defining an edge between two states. States are then parameterized by providing the specific prototypes used in each transition. There are several potential approaches to parameterizing a state model. These range from manual tools to more automated methods that passively observe interaction with interfaces. We currently use an authoring tool, leaving integration of more advanced methods for future work. Importantly, our methods separate modeling of state machines from recognition of widget appearance, allowing development of abstract models that can be used across a variety of concrete widgets.

At runtime the system maintains a set of parameterized state models that are potentially in progress. This set is populated using the active transitions monitored by Prefab. The triggering of a transition invokes a traversal across the corresponding edge in a state model. When a transition expires, its state model is given the option to expire, denoting a widget is no longer present in the interface. Abstract state models include setters for transitioning between states via manipulation of the interface. Specifically, the setter logic executes input events necessary for traversing edges in a state model. For example, the state model in Figure 5 packages a "click" function that sends mouse down and up events to the source application. This code is defined in the abstract state model, requiring only a single definition for any widgets that share this behavior.

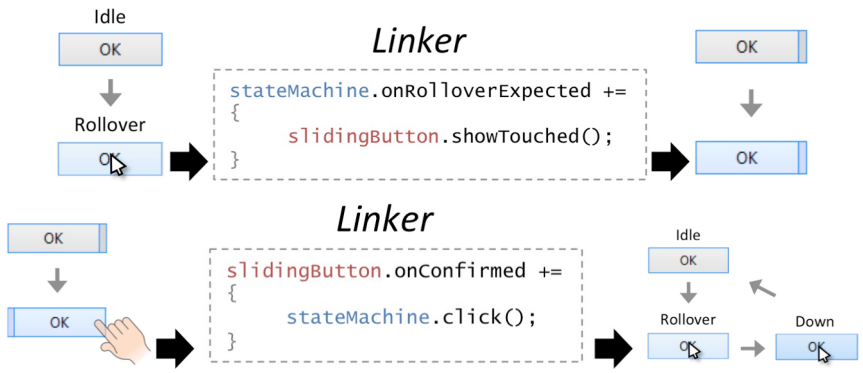


Figure 6.4: Linkers synchronize state models and Sliding Widgets. These two snippets are from a linker that synchronizes Sliding Buttons with mouse-based buttons.

Linkers

A state model defines the dynamics of an element, but those dynamics also need to be related to a Sliding Widget. We therefore provide *linkers*. Specifically, a linker (1) listens for edge traversals in a state machine and updates the Sliding Widget accordingly, and (2) listens for interaction events fired in a Sliding Widget and updates the state model. A single linker can be implemented for an abstract state model paired with a class of Sliding Widgets. For example, Figure 26 illustrates a linker that connects a button abstract state model to a Sliding Button, highlighting the flow of events passed between them.

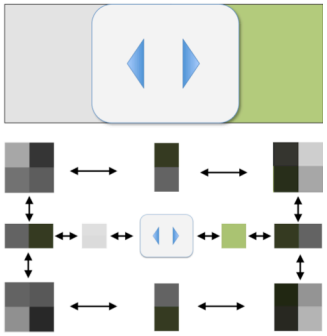
Upon observing an interaction with a Sliding Widget, a linker can induce arbitrarily complex state machine manipulations. For example, the linker in Figure 6.4 executes a range of actions depending on the observed Sliding Button interaction. The simplest case is when a person touches down on a Sliding Button and the linker induces a mouse rollover, traversing a single edge in the state machine. In contrast, confirming activation of a Sliding Button induces multiple edge traversals. The linker sends a click to the underlying button, which transitions it first to the state machine's pressed state and then back to the rollover state. Importantly, the separation of a linker from the Sliding Widget allows designers to create Sliding Widgets without a need to implement complex input redirection logic.

Although a state model can be manipulated by sending input events to the source interface, its state does not change until the underlying element's appearance changes. The length of this delay depends on time it takes to process input, and could be problematic for Sliding Widgets or other enhancements that want to provide instant visual feedback after an interaction. To address this problem, each active state model maintains pointers to two states: the *current* state and the *expected* state. The expected state reflects the state to which the machine is expected to traverse, but has not yet visualized. When a state model is manipulated via a setter, it first sets its expected state and then sends input to the underlying widget. A state model also fires an event when its expected state changes (e.g., onRolloverExpected in Figure 6.4), allowing a linker to provide instant updates to the Sliding Widget.

6.3 Mapping Widget Appearance

We have presented methods for modeling existing widgets, but there is also a challenge in styling the appearance of the runtime enhancement. Because our Sliding Widgets overlay existing interfaces, it is important that they maintain a style that is consistent with the underlying interface.

We address this problem with *mappers*, which automatically style Sliding Widgets consistent with the widgets they replace. In our solution, we first separate Sliding Widget style from content as in Hudson and Smith [34]. We also use customizable parts as in Hudson and Tanaka [35]. We then use mappers to translate Prefab prototype parts to the Sliding Widget parts. Prototypes model the pixel-level appearance of existing elements, and so we extend their usage from identification to also informing the rendering of new widgets.



Decomposition of a Sliding Widget into customizable parts is straightforward. Parts render fixed bitmaps or repeating patterns, such as simple repeating colors or more complex gradients. The pixels used in these parts are parameterized to define a specific appearance. Here a Sliding Button is decomposed into eleven parts, each parameterized with a simple appearance. Four bitmaps define the corners of the slider, four patterns the edges, two patterns the left and right troughs behind the thumb, and a single bitmap for the thumb itself. We use similar but more complex models to render other Sliding Widgets, such as the Sliding Spinners and Sliding Dropdown Menus in Figure 1.

Designers can leverage these decompositions to manually style Sliding Widgets, but our mappers expedite this process with methods for styling Sliding Widgets consistent with their underlying original interface. Mappers can be designed for reuse among many prototypes or for a specific prototype. They accept source prototypes and output Sliding Widgets with their parts parameterized based on those prototypes. Figure 6.4 illustrates three mappers currently in our framework: basic one-part mappers, nine-part mappers, and multi-prototype mappers.

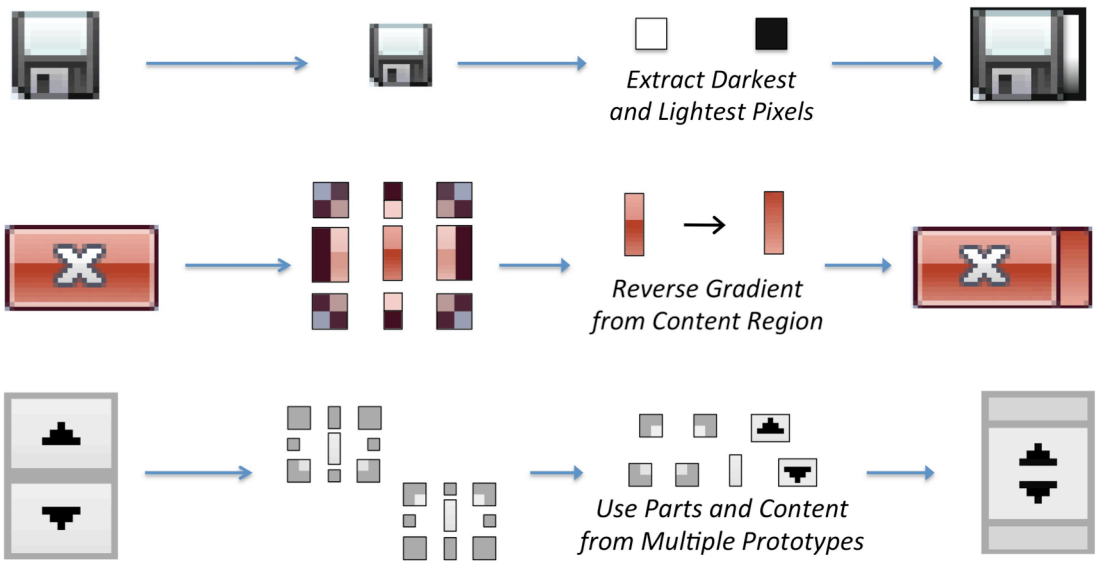


Figure 6.5: Mappers style sliding widgets consistent with the widgets they replace. Here are three mappers implemented in our framework. The top figure is a basic one-part mapper that translates icons into Sliding Widgets. The middle figure shows a nine-part mapper that maps the corners and edges of a button prototype to a Sliding Widget thumb. The bottom figure shows a multi-prototype mapper that combines content from multiple prototypes to render spinners.

One-part mappers translate one-part prototypes into Sliding Widgets. An example we have implemented uses two colors extracted from the prototype to define chrome in a Sliding Button. Specifically, we render the corners and edges of the Sliding Widget using the darkest color in the prototype. We then render the trough by extracting the lightest color in the prototype and creating a gradient from the darkest color to the lightest color. The top image in Figure 6.5 shows this one-part mapper applied to a save button.

Nine-part mappers obtain more complex renderings using Prefab's nine-part prototypes. The middle image of Figure 6.5 shows an example nine-part mapper that translates the appearance of a Windows 8 close button. It maps the corners and edges of the button prototype to the Sliding Widget thumb. It then obtains a recessed look by painting the trough using a reverse of the gradient obtained from the prototype content region.

The previous two examples use parts from a single prototype to parameterize the appearance of a sliding widget. It is also possible to perform more complex mappings using multiple prototypes. The bottom image of Figure 6.5 shows an example multi-prototype mapper where we map from prototypes that define the two arrows of a spinner. The content regions of those prototypes provide the up and down arrows, which are mapped into the thumb of the slider. Our associated video shows additional Sliding Widgets created using multi-prototype mappers, including a Sliding Checkbox created from two one-part prototypes that describe the appearance of a Windows 8 checkbox in its checked and unchecked states.

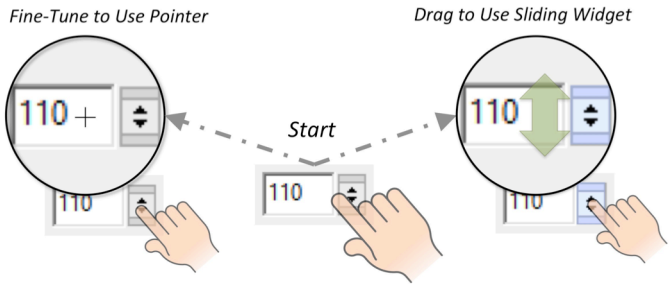


Figure 6.6: We modify Vogel and Baudisch's Shift [68], employing pointing when the user fine-tunes and contact area-based touches when the user drags.

6.4 Examining Sliding Widgets in the Wild

Sliding Widgets and other techniques addressing the fat finger problem are often designed, discussed, and evaluated using fields of abstract widgets or other limited testbeds. Because our implementation provides the ability to deploy Sliding Widgets, we sought to determine what insights we could gain from examining them in real-world interfaces. Our findings identify three challenges: (1) the challenge of supporting both area and point cursors within the same interface, (2) limitations of sliding in complex interfaces, and (3) limitations of replacing individual widgets.

Challenges with Supporting Area and Point Cursors

A major challenge is defining Sliding Widgets that work within the same interface as mouse-based widgets. Consider the spinner shown in the bottom-center of Figure 6.6, which illustrates a common scenario where a spinner controls the contents of an adjacent textbox. The spinner can be replaced with a Sliding Widget, but it then becomes difficult to manipulate the textbox directly. This is because Sliding Widgets leverage the entire contact area of touch input, but the touch here also overlaps the text field for which area cursor input is undefined. Sliding Widgets can resolve this ambiguity by alternating their orientation for disambiguation among other adjacent Sliding Widgets, but this strategy does not resolve the scenario where a touch area overlaps both a Sliding Widget and an element designed only for point-cursor interaction.

Sliding Widgets were unusable without the ability to disambiguate between pointing and sliding, so we developed a strategy for gracefully degrading from a contact area to a standard point cursor. More concretely, when a person touches a point-cursor element, we pass the standard touch point to the underlying interface (i.e, the contact area centroid). However, when a person touches a Sliding Widget, our overlay will send the entire contact area to the widget. In dense layouts, we employ an enhanced implementation of Vogel and Baudisch's Shift to enable fine cursor adjustment [68]. After a touch event, Shift creates a callout showing a copy of the occluded screen area and places it in a non-occluded location. Shift was designed for point-cursor interaction, so we extend it to support area cursor input. Our

goal is to make it possible to swiftly activate a Sliding Widget without eliminating precise control of a pointer. Thus, the core challenge in this task is to separate fine-tuning input from the dragging needed to activate a Sliding Widget.

Figure 6.6 storyboards our solution for disambiguating elements in dense layouts. When a person dwells over an element, we present a callout showing the occluded screen area. We then distinguish fine-tuning from dragging using a threshold on input velocity. Specifically, slow movements below the threshold are categorized as fine-tuning, and fast movements are treated as drags. Fine-tuning moves the position of a point cursor over mouse-based targets, and for any Sliding Widgets under the point cursor it sends touch down events. Alternatively, dragging only manipulates Sliding Widgets. The drag moves a Sliding Widget's thumb in the direction of the drag, allowing activation while hiding that input from any other point-cursor targets.

Limitations of Sliding in Complex Interfaces

Sliding Widgets use the metaphor of real-world sliding manipulations, but it is unclear how this applies to certain elements. One example is in hierarchical widgets, where an interactive element contains an interactive child element. The left of Figure 6.7 shows a screenshot of a panel in Adobe Photoshop where each row is clickable (i.e., to select a layer) along with its inner buttons (i.e., to toggle layer visibility). Unfortunately, this has no clear Sliding Widget analog. Sliding Widgets could potentially be nested in a larger Sliding Button, but this would be awkward when sliding either the outer or inner widgets.

We also found limitations of the sliding metaphor in the context of groups of related elements. Moscovich et al. demonstrate the promising feature of Sliding Widgets where a list of elements is selected in a single stroke. Our associated video also demonstrates an example of this interaction in the context of a Skype dialog. However, in most real interfaces where multiple elements are frequently toggled together, developers provide a "select all" toggle. In other cases, elements tend to be unrelated or infrequently toggled, leaving single stroke interactions undesirable.

This problem is largely due to conflicts between Sliding Widget behavior and the intentional design of interfaces for point cursors. More concretely, it may be possible to redesign an entire interface to exploit the advantages of single-stroke sliding interactions, but only at the obvious cost of needing to overhaul much of the design. Dixon et al. raised a similar tension between target-aware pointing and existing pointing-based interfaces [16], so perhaps this problem surfaces a larger conflict in designing surface-level modifications to existing designs.

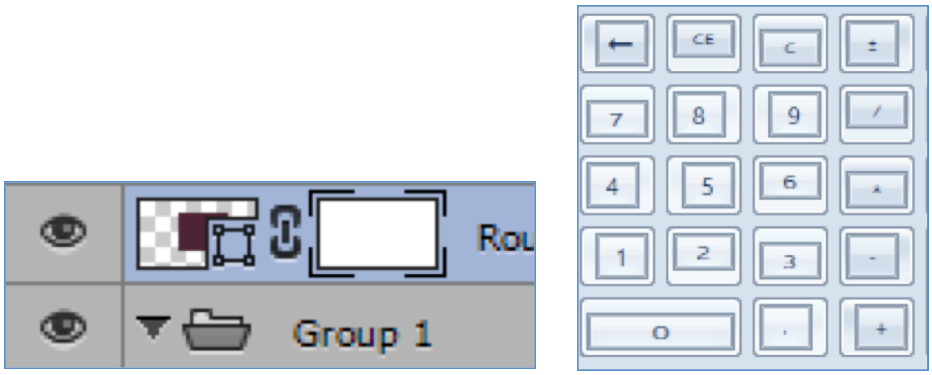


Figure 6.7: We identified limitations of sliding in complex interfaces and of replacing individual widgets. The left is a screenshot of a panel in Adobe Photoshop where each row is clickable along with its inner buttons. Unfortunately, this has no clear Sliding Widget analog. The right is an enhanced Windows Calculator interface. It includes a typical grid of buttons, but replacing each individual button yield an interface that is jarring and difficult to use.

Limitations of Individual Widget Replacement

Our implementation mostly replaces individual elements with individual Sliding Widgets. However, this approach can introduce conflicts when multiple widgets are related.

We partially anticipated this in our implementation of a Sliding Spinner. A naïve implementation would replace each button in the spinner with a Sliding Button and alternate their direction to help with disambiguation. However, this can conflict with a person's expectations of the interface if the chosen sliding direction is different than the semantic direction of the spinner. For example, the spinner in Figure 6.6 adjusts the numerical value in its textbox, and sliding horizontally may seem less appropriate than sliding vertically. The problem is magnified if the arrows from the original spinner are mapped into the appearance of the Sliding Widget. We addressed this problem with the Sliding Spinner that understands the relation between the two buttons and behaves appropriately.

While our strategy for grouping multiple related widgets works for spinners, the strategy is much more difficult to implement for larger collections of related widgets. For example, consider the enhanced Windows Calculator interface shown in the right of Figure 6.7. It includes a typical grid of buttons, but replacing each individual button yields an interface that is jarring and difficult to use. The calculator's original layout is relatively dense, so it seems useful to alternate sliding directions. But then a person has to carefully inspect each button to understand how to slide it. Instead, the entire panel of buttons should be replaced with a Sliding Widgets optimized for these types of grid layouts.

This problem suggests that a more global understanding of interface layout is necessary for replacing some elements. One possible strategy is to employ a mechanism similar to Nichols et al.'s Smart Templates for replacing entire groups of controls [52]. This technique uses parameterized templates to specify when different conventions might be applied. Our approach to replacing spinner widgets can be

seen as an initial example of this, but we imagine there is an opportunity to extend this idea to more sophisticated templates capturing a variety of conventions and semantics.

6.5 Discussion

Although we describe our pixel-based methods in the context of implementing Sliding Widgets, our modeling of state and our approaches to styling widgets using prototype parts can enable and inform a variety of future enhancements. For example, other enhancements can directly re-use our state models. An enhancement to support crossing-based widgets on hybrid pen and mouse devices could use our same state models, implement new linkers relating their crossing widgets to our models, and style their widgets using our techniques. Even more complex enhancements might go beyond replacing individual widgets, using our models to drive entire new interfaces automatically generated by a system like Supple [27].

We described how our state models provide getters and setters for widget state, encapsulating low-level details required for monitoring and manipulating widgets. These turned out to be additionally beneficial because they provide an API that is similar to the API of the underlying toolkit itself. For example, our checkbox state model provides a familiar getChecked() method, which hides details of what states and prototypes the model uses to represent the checkbox. We believe that there is a rich opportunity for future work that investigates a more general set of *semantic views* that encapsulate pixel-based methods. The goal of these views would be to provide developers of pixel-based enhancements with a higher-level toolkit that is more similar to existing interface toolkits. For example, a getLabel() method on a semantic view of a button would manage the details of obtaining the button's prototype, recovering its content, and processing those pixels to recover its text.

Our linkers automatically map Sliding Widget interactions into a sequence of mouse events to manipulate the underlying interface element. Current windowing systems are not designed to support this type of behavior. In particular, the overlay and the underlying enhancement must share the same input stream, which requires blocking input from one or the other during different stages of an interaction. This limitation also makes it difficult to support simultaneous manipulation of elements (e.g., Moscovich et al. suggest using an area cursor to drag two sliders at the same time). Deeper exploration of new input management frameworks is an opportunity for future work. For example, a framework might provide multiple input streams, separating live input from redirected input.

We implemented several basic linkers for synchronizing Sliding Widgets and state machines in most interfaces, but we imagine developers will extend these or create more sophisticated linkers tailored for specific applications. As one motivation, our example linker maps touches to mouse rollovers, thus

making it possible to view tooltips or other rollover responses in existing interfaces. But Moscovich et al. also demonstrated more complex responses to touch events with their action-preview-on-touch behaviors [47]. In their design, touching a button displays a preview of the action it will perform if activated. Future work might therefore examine linkers that execute these previews by monitoring and manipulating multiple state models. Importantly, our Model-View-Controller pattern for separating linkers from state models and Sliding Widgets makes it possible to define custom behaviors without modifying state model or Sliding Widget code.

Our application is the first practical general-purpose implementation of Sliding Widgets. There is nearly always a fundamental gap between the knowledge that can be gained in lab studies versus the implications of this knowledge for real-world contexts, and we believe this work narrows the gap for Sliding Widgets. We are exploring the best way to deploy our application on Windows 8 hybrid devices to further bridge the gap from the lab to the field. We also envision tools accompanying the deployment where developers can gather logged usage data to provide insight into real-world challenges (e.g., testing alternative designs, visualizing problems reported by end-users of pixel-based enhancements). Our hope is to help catalyze interaction research in escaping the lab, putting it into the hands of end-users who stand to benefit from the field's rich innovation.

Chapter 7 | **LAYERS AND ANNOTATIONS**

Pixel-based methods can enable modification of interfaces without their code, independent of their implementation. For example, the applications we explore in the previous chapters use pixel-based methods to enhance the entire desktop. Chapter 3 presents an implementation of the Bubble Cursor [16,28], which dynamically resizes to always select the nearest target. Chapter 6 describes our implementation of sliding widgets [18,47], which replaces mouse-based interface elements with touchscreen widgets. Chapter 4 also presents our enhancement that translates the language of interfaces for improved localization [19]. Other examples include accessibility enhancements, testing frameworks, automation tools, and help systems [17,19,74,75]. These enhancements modify interfaces in a variety ways, but they are all enabled by methods that use pixels as a universal representation.

Unfortunately, enhancements like these are difficult to implement, and so relatively few are available. There are at least two major reasons for this. First, interpreting an interface from its raw pixel values is a large and multi-faceted problem. For example, the translation enhancement requires mechanisms to identify interface elements, recover text, and perform higher-level analysis of that text. Requiring all of this functionality in a single application quickly leads to monolithic code that is difficult to develop and maintain. This problem is magnified by the fact that different enhancements require different interpretations. For example, in contrast to the translation enhancement, the Bubble Cursor is agnostic to text values and only needs to identify clickable targets. When enhancements do require similar methods that could potentially be reused, the current lack of structured support leads developers to re-implement large portions of code.

Second, writing sophisticated code is not enough to successfully interpret most interfaces. This is because some information is not obtainable through raw pixel analysis. For example, Dixon et al. report that clickable "targets" are often ambiguous and cannot be reverse engineered without human intervention [16]. This ambiguity is pervasive in pixel-based methods and causes unavoidable errors. Developers then patch their code by mixing in their own outside knowledge about an interface. This results in fragile and monolithic code that is difficult to broadly deploy.

We address these two problems in a new toolkit for pixel-based reverse engineering of graphical interfaces. First, *Prefab Layers* helps developers write interpretation logic that can be composed, reused, and shared to manage the multi-faceted nature of pixel-based methods. Second, *Prefab Annotations* supports robust annotation of interface elements with metadata that has been inferred, provided by a

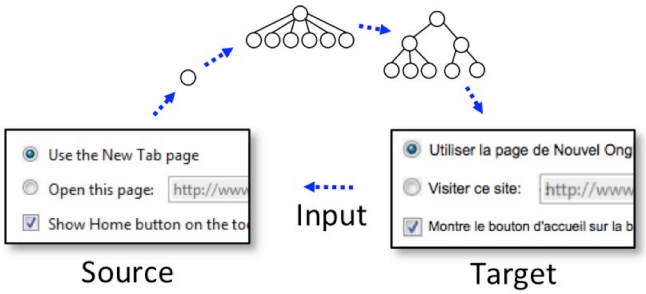


Figure 7.1: Prefab Layers and Prefab Annotations structures pixel-based interpretation as a series of tree transformations.

developer, or collected from end-users of pixel-based enhancements. Together, Prefab Layers and Prefab Annotations help developers focus on the high-level functionality of their enhancements instead of the low-level challenges of pixel-based analysis.

Contributions of this chapter to pixel-based methods include:

- Prefab Layers, which offer a set of techniques to simplify the development of interpretation code while encouraging composition and reuse.
- Prefab Annotations, which offer a set of techniques for managing portable and robust metadata.
- Demonstration of reusable interpretations implemented using our methods. These range from low-level reusable components to high-level pixel-based enhancements.
- A comparison of how developers use our methods versus state-of-the-art tools. We show that our methods speed up development time, enable code reuse, require less data management, and help developers focus on the high-level behavior of their enhancement.

7.1 Overview and Scenario Walkthrough

We now introduce our toolkit with a brief overview. To clarify how developers use our toolkit, this section also presents a scenario that walks through the development of the interface language translation enhancement in Figure 7.2.

Toolkit Overview

Prefab Layers and Prefab Annotations decompose interpretation into a series of tree transformations, as shown at the top of Figure 7.1. Interfaces are hierarchies and can be reverse engineered by iteratively working from raw pixels to a detailed interpretation. Developers implement custom interpretations for their enhancement using a combination of layers, layer chains, interface metadata (specifically tags and annotations), and annotation libraries.

A **layer** is a script that performs a specific set of tree transformations using the current structure and properties of the interface hierarchy, the pixel values of the captured screenshot, and any interface metadata.

A **layer chain** is a group of layers that execute in sequence. An interface is interpreted by passing the raw image into the first layer as a single root node, then passing the output of each layer into the next. Developers reuse and compose existing functionality by concatenating layer chains. Alternatively, they modify or enhance a chain by adding or replacing layers. As mentioned, Prefab recovers a spatial hierarchy from pixels, and so we include this functionality as the default set of layers at the beginning of a chain. Developers typically append their own custom layers to infer higher-level semantics on top of Prefab's hierarchy.

Interface metadata stores information about a specific node in a hierarchy (e.g., whether a node is a target, a corrected translation for a given node). A critical distinction is the intended persistence of the metadata representation. A **tag** is interface metadata stored on a node in the hierarchy created by a particular layer chain's interpretation of a source image. An **annotation** is interface metadata described in terms of the source image, which can be persistently stored and used with different layer chains.

An **annotation library** is a set of related annotations, and a developer will typically create a library for each type of annotation used by an enhancement. Before a layer chain executes, each layer can *import* annotations to be used at runtime. The details of how these are robustly stored and imported are challenging and discussed in a later section.

Enhancements are therefore implemented by creating and composing combinations of layers and annotation libraries. An enhancement might use several instances of a layer, but point each at a different annotation library. Alternatively, an enhancement might contain several layers that work together and share a single annotation library.

This toolkit is implemented in C#, with annotation libraries implemented as CouchDB databases. We selected CouchDB in part because its replication support allows easy sharing and synchronization of annotation libraries. As a convenience, we also provide an interpreter that uses Iron Python to allow layers to be defined in short Python scripts**.** For clarity, this chapter presents its example layers in Python.

Scenario: Runtime Interface Translation

To better understand how developers use our toolkit, let us follow Emily as she implements the interface translation enhancement from Figure 7.1. Emily has found many applications do not support her native

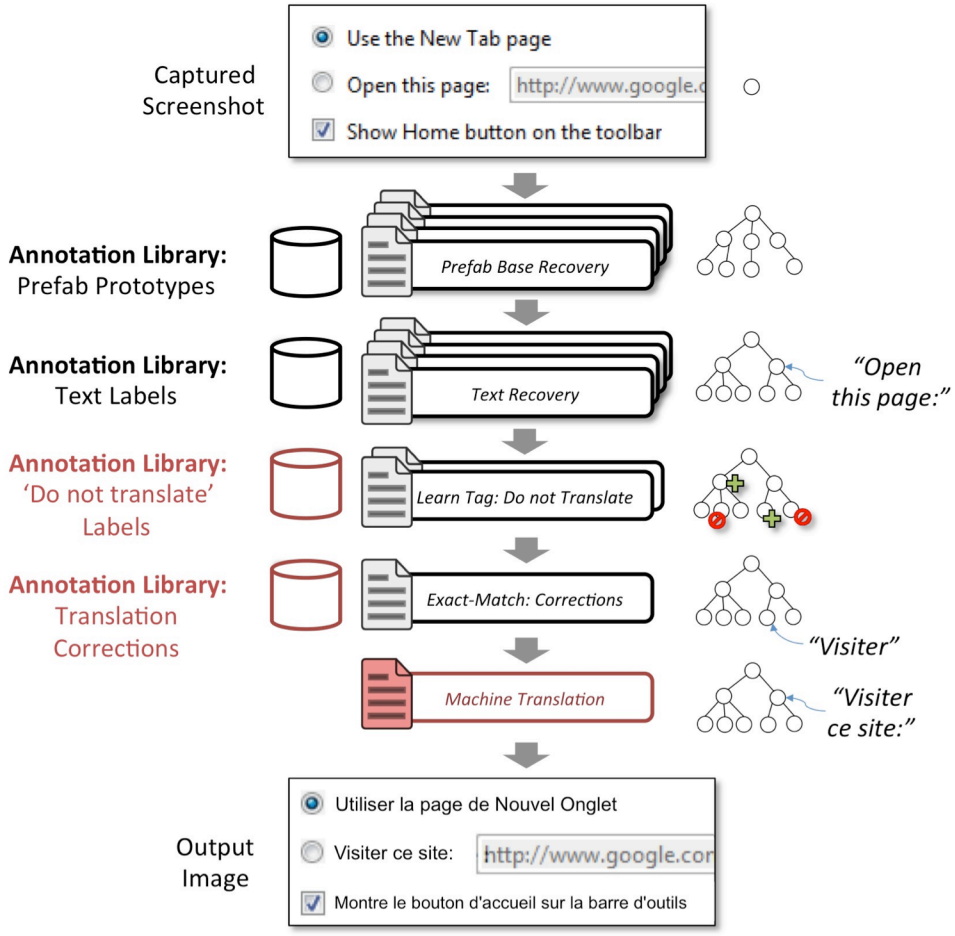


Figure 7.2: My toolkit simplifies runtime interface translation. It uses layers that recover interface text, decide what text should be translated, and then present translations obtained using both machine translation and human correction.

language, or their translations are erroneous and incomplete. Instead of being stuck hoping application developers will add or fix their translations, she decides to develop an enhancement to recover text from the pixels of an interface, translate that text, and re-render interfaces incorporating the translations. Figure 7.2 illustrates her implementation. Emily primarily composes and parameterizes existing layers, writing only a small amount of custom behavior (as shown here in red).

Emily starts by creating an input and output redirection loop as shown in Figure 7.1, within which a layer chain executes upon receiving a screenshot. She then imports a layer providing Prefab's base recovery capabilities. This outputs a tree representing the elements of an interface, as recovered from its pixels. Emily finds the base layer does not recover text (an optimization based on the fact that text recovery is expensive and many enhancements do not require textual content). Emily will obviously need the interface text, so she adds a standard text recovery layer. The first two lines of Figure 7.3 show the implementation of this functionality.

```
# main.py
params = { 'library' : 'translation_corrections' }
prefab_layers.import_layer('apply_annotations', params)

import prefab_layers
chain = prefab_layers.new_chain()
chain.import_layer('prefab_identification')
chain.import_layer('text_recovery')



# translate_text.py
from microsoft_translator import Translator
translator = Translator('client id', 'client secret')

def interpret(interpret_data):
    ''' This method is called by the Prefab Layers
        toolkit when it needs this layer
        to perform its transformations '''
    root = interpret_data.tree
    recursively_translate (root, interpret_data)

def recursively_translate(currnode, interpret_data):
    ''' This method recursively visits each node
        and translates its text value to French '''
    if currnode.get_tag('is_text'):
       text = currnode.get_tag('text_value')
       french = translator.translate(text, 'fr')
       interpret_data.add_tag(text,'translation',french)

    for child in currnode.get_children():
       recursively_translate(child, interpret_data)
```

Figure 7.3: This code demonstrates how to translate the language of an interface using Prefab Layers and Prefab Annotations. The main.py file at the top describes how to import layers and provide them with an annotation library. The translate_textl.py file implements the functionality for translating nodes in the hierarchy that have been tagged with text.

Emily is unable to find a layer implementing translation, so she authors a custom layer. Within her layer, she walks the tree recovered by previous layers. For each node containing recovered text, she runs the text through a web translation service and tags the node with the resulting translation. The translate_text.py layer shown in Figure 7.3 implements this behavior.

With her core layers built, Emily now focuses on the behavior of her enhancement. For each node tagged with a translation, she uses Prefab's pixel-level methods to overlay a mask that removes the element's original text. She then renders the translated text within the same bounds.

With her enhancement now working, Emily finds many of the translations are erroneous and decides to add interactive correction. She builds an interface that allows a person who observes an erroneous translation to view the original text and provide a correction. She wants these corrections to be persistent, so she stores them as an annotation library. She then imports a layer that uses the annotations at runtime to tag elements with their corresponding corrections. The remaining lines of main.py in Figure 7.3 implement this behavior.

Emily also notices she is translating text that should not be modified, such as system paths in file widgets. Emily adds an annotation library for a "do not translate" flag, updates her interface to allow toggling this flag for any element, adds a layer to tags nodes according to this annotation, and updates her machine translation layer to respect the flag.

Emily is satisfied her enhancement gives control over whether to translate each element, but wants to minimize the need to tag elements. She therefore uses a layer that trains a classifier using collected annotations as training data. Emily does not need to implement this functionality, she just imports an existing layer and parameterizes it to use her annotation library for training. It then learns a classifier based on the annotations and at runtime tags any nodes that the classifier determines should not be translated.

With her enhancement implemented, Emily uses it in a variety of applications. She can also share her enhancement and its annotation libraries with other people. Eventually, she might decide to parameterize her layers to allow people to choose a target language. This would require adjustments to how she invokes machine translation and would probably introduce different annotation libraries to store corrections in different languages. Importantly, her layers and annotations continue to work together, so she is not burdened with migrating data or code as she iterates.

7.2 Prefab Layers

In defining pixel-based interpretation as a series of tree transformations, the primary challenge is ensuring layers have the power and flexibility to construct arbitrary interpretations while also preserving simplicity in each layer (i.e., obtaining a high *ceiling* and low *threshold* [48]). The naïve approach of simply allowing layers to arbitrarily mutate a hierarchy falls short for at least two reasons. First, it could be expected to easily regress to monolithic layers, undermining our goals for reuse and composition. Second, we have found it difficult to reason about the entire structure of a 2D interface, especially when in-progress mutations mean the current hierarchy represents neither the input nor the output of a layer. It

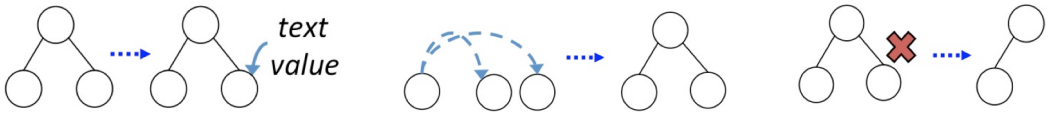


Figure 7.4: Layers can execute three kinds of operations to transform a hierarchy: tagging a node, setting an ancestor for a node, and deleting a node. These options were chosen for simplicity and completeness. Setting an ancestor, for example, allows developers to add many nodes to the hierarchy without having to manually reorganize the entire structure.

is difficult to even traverse a hierarchy while also mutating it, and we found more complex transformations near impossible to reason about.

Prefab Layers therefore gives each layer an *immutable* view on its input. We provide a set of tree transformation operations, and layers can request any number of operations be applied to nodes in the hierarchy. All operations are then applied in batch after the layer terminates (i.e., the hierarchy is mutated *between* layers in a layer chain). This guarantees layers always observe trees that are in a stable state, making it easier to reason about. It also limits the scale of transformation that can be accomplished in a single layer, encouraging developers to think of interpretation in discrete steps. In the terminology of Myers et al. [48], we create a *path of least resistance* toward reuse and composition by leading developers to create layers that each implement a single piece of well-defined functionality.

The specific operations are shown in Figure 7.4: *tagging* a node, *setting an ancestor* for a node, and *deleting* a node. These options were chosen for simplicity and completeness. Each operates on a single element or a pair of elements, and they can be combined to create any hierarchy.

Tagging. Layers can add interface metadata at runtime by tagging nodes. Subsequent layers can read that metadata to inform their own execution.

Setting an Ancestor. Layers can require a hierarchy be modified to ensure a given node is an ancestor of another given node. This can be set for two existing nodes (i.e. set one as the parent of the other), for an existing ancestor node (i.e., inserting a new child), or for an existing child (i.e., inserting a new parent).

Deleting. Layers can delete nodes from a hierarchy. Any children of a deleted node are attached to the deleted node's parent.

After a layer requests a set of operations, we efficiently apply all operations to its input hierarchy. Tag operations are trivial, but are executed in batch as part of encouraging layers that perform a single step of interpretation. For each delete operation, we remove the node and attach its children to the deleted node's parent. We apply ancestry requests by adding an edge from each ancestor to its descendent and

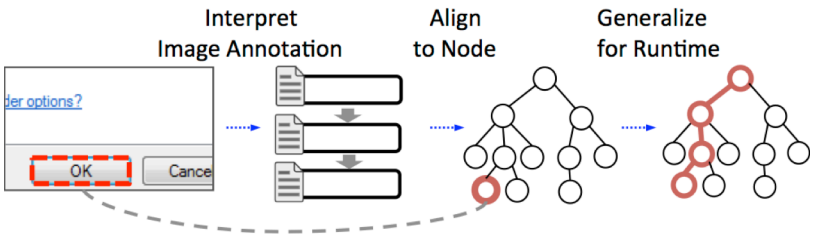


Figure 7.5: When a layer imports an annotation library, we provide a tree representation for each annotated element. Image annotations are interpreted by preceding layers, aligned to nodes in the resulting hierarchies, and then handed to the layer, where it computes information it will need at runtime.

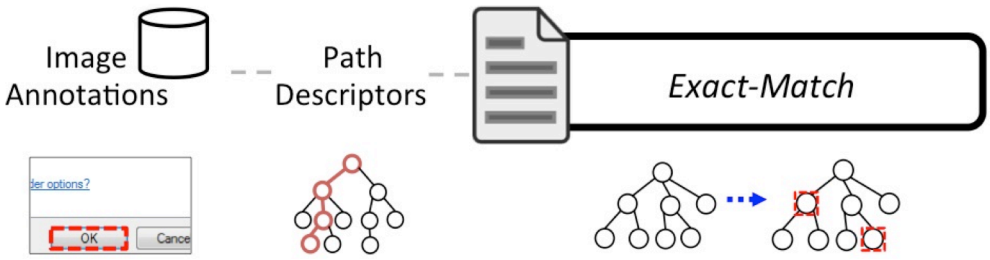
pruning any redundant edges. Finally, we raise an exception if these requests do not produce a valid tree (e.g., if operations create cycles or multiple paths between nodes).

7.3 Prefab Annotations

The goal of Prefab Annotations is to store metadata about specific interface elements, such that the metadata can be accessed and shared by arbitrary layers. Storing metadata is trivial, but robustly storing the interface element itself is challenging. This is because different layer chains represent the element differently (i.e., the tree structure recovered from a screenshot depends on the specific layer chain used). For example, the sliding widgets enhancement requires a tree with related buttons grouped (so they can be replaced with a single slider, as in Figure 1.4). In contrast, language translation can leave these buttons separate, but needs to group related text. As a result, annotations cannot be stored in an encoding that depends on the structure of one specific layer chain. Otherwise it would be difficult to share libraries of annotations among enhancements. Even within a single enhancement, a developer would not be able to iteratively build and test a layer chain because its tree representations would change throughout development.

Prefab Annotations address this challenge by storing annotations using a *pixel-level* representation. Specifically, an annotation library contains a set of image annotations, each stored as an image of an interface, a region within that image to be annotated, and associated metadata. When a layer imports an annotation library at runtime, the pixel-level representations are converted to trees consistent with the current layer chain. Therefore, the sliding widget enhancement views an annotated element as a node in a tree where related buttons are grouped together. Similarly, the translation enhancement views that same annotated element in a different tree where text is grouped together.

Figure 7.5 illustrates this process. First, images in the library are interpreted by the *preceding* layers in the current chain. This creates hierarchies consistent with the current runtime. We then use region information in each annotation to identify the corresponding hierarchy node. We pass these matched pairs of image annotations and aligned tree nodes to the layer (together with a list of image annotations that do




```
def import_annotations(annotation_data):
    ''' This method indexes annotations using
        path descriptors, so it can tag these
        annotated elements at runtime.'''
    runtime_storage = annotation_data.runtime_storage
    annotated_nodes = annotation_data.annotated_nodes
    for annotation in annotated_nodes:
        path = get_path(annotation.node, annotaion.root)
        runtime_storage[path] = annotation.metadata

def interpret(interpret_data):
    ''' This method tags nodes with annotation metadata
        by matching path descriptors against each node '''
    currnode = interpret_data.node
    path = get_path(currnode, interpret_data.tree)
    if path in interpret_data.runtime_storage:
        metadata = interpret_data.runtime_storage[path]
        for key in metadata:
          interpret_data.add_tag(currnode, key, metadata[key])
    #recurse on children
    for child in currnode.get_children():
        interpret(interpret_data, child)
```

Figure 7.6: Layers import annotations for use at runtime. Here an *exact-match* creates a path descriptor from each annotation and uses those descriptors to tag nodes at runtime.

not match any nodes). Using these matched pairs, the layer computes and stores any information it will need at runtime (e.g., path descriptors, a learned classifier).

For clarity, Figure 7.6 presents the entire script for a simple layer that uses annotations. This layer tags elements at runtime that were previously annotated with metadata. The layer indexes each annotation using a unique reference that can be matched against nodes at runtime. Specifically, it computes an XPath-like *path descriptor* based on properties of the annotated node and its ancestors. Importantly, this layer does not attempt to generalize the annotation. At runtime it visits each node in the input tree and checks if the node's path matches any of descriptors it has stored. If there is a matching descriptor, it tags that node with the corresponding metadata. These path descriptors always represent a valid path that can be retrieved at runtime because they are computed from hierarchies consistent with the current layer chain. For example, the sliding widgets layer can import this layer and use it with an annotation library of "replace this widget" Boolean flags. The language translation enhancement can similarly import this layer with a library of translation corrections.

This simple tagging layer is one of several *exact-match* layers in our toolkit, each performing one of the tree operations in our toolkit. An exact-match deletion layer similarly deletes any node that matches a

path descriptor. These layers are designed to be simple building blocks for more advanced layer chains, and the next section introduces several additional strategies that go beyond exact-matching of path descriptors to more sophisticated generalizations.

7.4 Validation through Example Layer Chains

Our toolkit is designed to support diverse pixel-based methods. In this half of our validation, we demonstrate and give insight into our toolkit by implementing and discussing several example layer chains. In the terminology established by Olsen [55], these examples demonstrate an *inductive combination* of functionality. Specifically, we select examples to illustrate how interpretation code can be composed, reused, and shared, and also how annotations can be used and shared among enhancements. We start with examples of low-level reusable interpretations and then move to high-level composition in full enhancements.

Reusable Low-Level Layer Chains

This section expands upon the previous section's reusable exact-match layer. Specifically, we present two more example layer chains: (1) a reusable chain that learns to automatically generalize Boolean annotations, and (2) a text recovery chain that applies multiple techniques to recover text from an interface. Like the exact-match layer, these are reusable building blocks that can be parameterized with an annotation library to obtain a desired capability. Decoupling the code layers from the data annotations thus creates building blocks for developers to create complex behaviors.

Learning-Based Annotation

Exact-match annotations are sufficient and even preferable for many applications, but others benefit from expediting annotation through generalization. We support such inference with a layer chain that learns to tag nodes based on positive and negative example annotations. For example, Figure 7.2's demonstration uses this chain to generalize its "do not translate" annotation. We implement learning as two layers sharing a library of Boolean annotations. The first is an exact-match layer, tagging nodes that are explicitly annotated as either positive or negative. The second applies a classifier to generalize tags onto nodes that are not explicitly tagged. Annotations are thus treated as ground truth and always override the classifier. Sharing the annotation library means a single annotation both tags a specific element and contributes to training the classifier.

Our learning layer uses a decision tree, with features computed from an element's spatial properties, its location in a hierarchy, and tags applied by preceding layers. It imports annotations by using them as training examples to create a classifier. During interpretation, it applies the classifier to nodes not tagged

by the exact-match layer. Our current classification algorithm was designed and evaluated in the context of the applications described in this chapter, so its performance is optimized for our explorations. However, our goal in implementing this example is to illustrate how any learning algorithm could be deployed in our toolkit. Developers could swap in layers that implement custom classification algorithms tailored for their application, or could use general-purpose classifiers they customize by populating an annotation library of training data.

Text Recovery

The exact-match and learning-based chains are relatively simple, with their power coming from how they can be composed. But it is also possible to implement complex layer chains providing sophisticated reusable functionality. One example is our current chain to recover textual content. Prior work has found the extremely low resolution of interface text makes it difficult to implement text recovery with off-the-shelf character recognition, instead turning to human transcription [19]. Other work explores incorporating text from the accessibility API [11]. Failures are inevitable in both approaches, so we leverage the flexibility of our toolkit to combine recovery from the accessibility API with human transcription. Figure 6 presents an overview of the layer chain, which consists of four main components: text classification, grouping related text, human transcription, and accessibility recovery.

The first layer tags each interface element with a Boolean indication of whether the node represents text. The chain is a parameterization of the learning-based annotation chain, trained with positive and negative examples of text elements. Prefab's background differencing discovers many types of elements (e.g., text, icons, widgets), but does not indicate the types of those discovered elements. This layer therefore identifies which should be processed as text.

The second and third layers are used to group related text. Low-level methods often naturally group text within a parent. For example, a button with a two-word label will group the two text elements. But text rendered without a visible container may need explicit grouping (e.g., a checkbox may have a multi-word label with no visible enclosure). We implement grouping in two layers. A first learns to tag each element with a Boolean flag indicating whether it should be grouped with the next sibling in reading order. A second then performs the actual grouping.

The final two layers tag elements with text values. The first obtains these from human transcriptions. It works similar to an exact-match layer, but generalizes the annotations differently. It uses a hash of the pixel-level appearance of annotated text to apply that same annotation whenever it finds the same pixels (e.g., it matches multiple buttons with the same "OK" label). The second layer obtains text values from

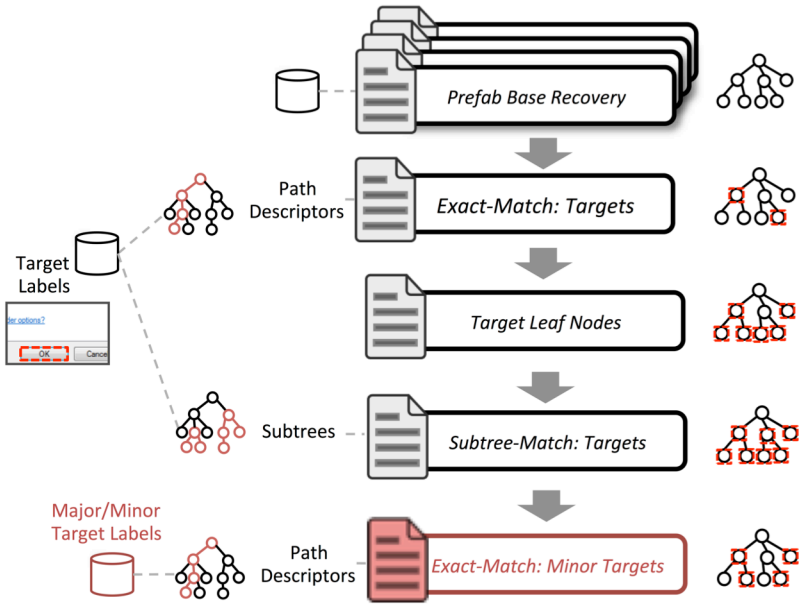


Figure 7.6: Prefab Layers and Prefab Annotations can be used to streamline and expand the general-purpose Bubble Cursor enhancement [16]**, resolving important limitations of our previous implementation.**

the accessibility API hierarchy corresponding to the captured interface. It finds the set of corresponding nodes in the pixel-based hierarchy by testing for spatial containment, and tags the parent node with the textual labels.

Full Layer Chains for Enhancements

Our final examples demonstrate how our toolkit can be used to compose full layer chains used by enhancements. Specifically, we examine interface translation, target-aware pointing, and sliding widgets. Our toolkit both: (1) lowers the *threshold* to developing enhancements, and (2) raises the *ceiling* to enable new pixel-based enhancements.

Language Translation

We previously presented pixel-based language translation [19], but the implementation in that initial work is fragile. Lacking explicit support for modular and reusable code, most of that work focuses on text recovery methods and ignores important aspects of translation. For example, that work ignores interactive correction (e.g., incorrect machine translations, elements that should not be translated at all). Figure 1.3 and our introductory scenario present a new and more extensible implementation addressing these issues. My new solution builds directly on reusable low-level layer chains, so there is minimal new work required (as illustrated by Figure 3's red highlights of that new work).

Target-Aware Pointing

My prior Bubble Cursor implementation is an example of the state-of-the-art in pixel-based enhancements [16,28]. It enables target-aware pointing across the entire desktop, identifying targets using Prefab and

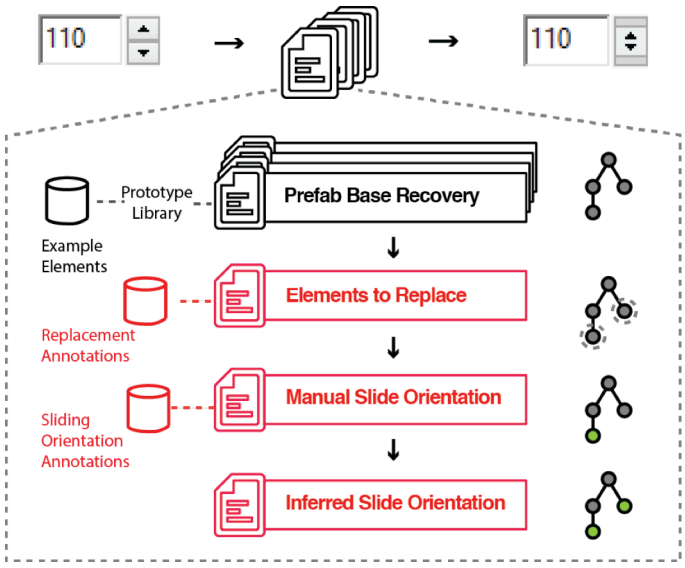


Figure 7.7: Prefab uses examples of interface elements to generalize prototypes of the appearance of families of widgets. Building from a library of these prototypes, we interpret which elements to replace with Sliding Widgets as well as their sliding orientation. Our novel methods are shown in red, while Prefab's existing methods are in black.

human annotations. Specifically, it: (1) uses Prefab to identify a hierarchy of elements, (2) uses path descriptors to store annotations about whether to target an element (e.g., target a button, do not target an icon), and (3) expedites annotation with two heuristics, one that targets leaf nodes and one that generalizes annotations across identical subtrees.

My new toolkit dramatically streamlines this enhancement. Figure 7.6 shows our new implementation, where we first instantiate a base Prefab layer and then an exact-match layer for target annotations. We then add two custom layers, one for each of the target heuristics. Importantly, the subtree matching layer shares the same annotation library as the exact-match layer, but generalizes those annotations. Alternatively, we could replace these lightweight layers with our learning-based chain from the previous subsection.

We also expand upon our prior work by differentiating *major* and *minor* targets. In our prior work we identified that some targets are infrequently used and act as distractors to more important targets. We suggested it would be valuable to distinguish major versus minor targets, but our prior monolithic implementation did not allow exploration of this idea. Figure 7.6 shows that our new toolkit makes it trivial to add a layer and annotation library to support exploration of this distinction. Existing target annotations can continue to be used, and a new annotation is added to indicate which of those are minor targets. This example shows our new abstractions raising the ceiling relative to what we could accomplish with the prior state-of-the-art.

Sliding Widgets

The previous two layers streamline the implementation of the Bubble Cursor and Language Translation, but our toolkit also lowers the threshold to implementing Chapter 4's sliding widget enhancement. The sliding widgets enhancement is tedious and impractical to develop without the toolkit. Developing a new enhancement like this requires iteratively refining code and exploring different approaches. But that code also requires annotations, and prior approaches to storing such annotations are brittle to code changes and would need to constantly be migrated. My toolkit also provides a clear conceptual model for reasoning about and describing our sliding widgets enhancement. Figure 7.7 presents a figure similar to the layer chain illustrations throughout this chapter, succinctly overviewing the code and data used in the sliding widgets enhancement. My current contribution is therefore not only code, but also a set of abstractions that allow effective description of pixel-based interpretation.

7.5 Validation with Developers in the Lab

The previous section explored examples implemented using our toolkit. We also conducted a lab study that compared how developers used our toolkit versus a baseline toolset.

The baseline tools were designed to reflect state-of-the-art methods presented in our prior implementation of the Bubble Cursor [16]. Specifically, there were two main differences between our toolkit and prior methods. First, the baseline interpretation logic is implemented as a single method, where developers manually organize any code invoked in that method. Second, the baseline stores annotations using path descriptors, as used in our prior work and in a wide variety of DOM-based enhancements.

To the best of our knowledge, Prefab's prior methods provide the most extensive support for combining code and data in real-time modifications. Yeh et al.'s Sikuli offers robust scripting tools [74], but less support for annotating arbitrary interface elements. In addition, their pixel-based methods require a reported 200msec to identify all occurrences of a *single* target. My toolkit is designed to support enhancements that need to more quickly identify and annotate all occurrences of *many* elements, so Prefab's prior methods are a more meaningful comparison.

We recruited six experienced developers to participate in our study. They were all male, with ages ranging from 24 to 28 years. Although all currently develop software, their backgrounds included applied natural science, computer vision, software engineering, and programming languages. All were familiar with Python and at least one other programming language. None of the participants had experience with pixel-based reverse engineering.

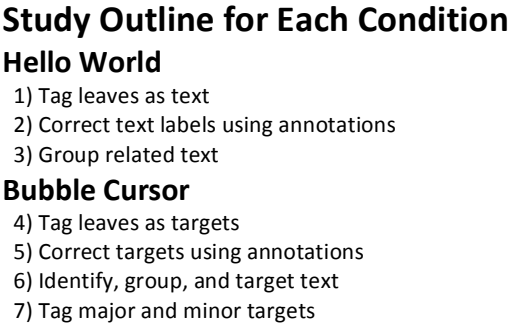


Figure 7.8: Participants were given 7 tasks in each condition. The first three implement a *Hello World* enhancement. The next four implement and expand upon the *Bubble Cursor*.

Study Protocol

Study sessions took approximately three hours. Participants were asked to implement interpretation logic for two enhancements: *Hello World* and *Bubble Cursor*. Both were implemented twice, once with our toolkit and once with the baseline. Participants were allotted 90 minutes for each task. To control for learning effects or fatigue, we counterbalanced the order in which we presented each condition. The *Hello World* enhancement was designed to familiarize participants with the condition. It replaces identified text with the string "hello world", using the masking technique described in our scenario walkthrough. In both enhancements, participants were asked to reverse engineer screenshots captured from an Apple iTunes dialog and a Microsoft Word settings dialog.

The experiment was conducted using a standard desktop computer running Windows 7. Developers authored code using an off-the-shelf text editor. We also equipped participants with a debugging tool that worked similarly to a webpage DOM inspector. Using the tool, participants could load screenshots, compile and execute their code, navigate and inspect the hierarchy output by their code, add or delete annotations, and browse through their annotations.

Both conditions consisted of seven tasks, as in Figure 7.8. The first three were to identify text for the *Hello World* enhancement. Participants implemented a simplified version of text recovery, where (1) leaves are heuristically classified as text, (2) erroneous tags are corrected with annotations, and (3) related text is grouped. For the grouping task, we wanted to examine how developers make use of existing code, so we gave participants access to grouping logic originally developed for an enhancement that classified widget types. In our toolkit, the functionality could be imported as a layer chain. In the baseline, it could be obtained using Python's standard mechanism for importing modules. Developers were also free to copy and paste any code from the existing enhancement.

The next four tasks were to classify targets for the *Bubble Cursor*. In these tasks, participants were able to reuse any functionality from their *Hello World* enhancement. Two tasks directly correspond to our original implementation: (4) leaf nodes are tagged as targets, and (5) erroneous tags are corrected with annotations. The next then improved upon that implementation by (6) identifying and grouping related text so that it can be targeted. We have noticed that groups of text are often clickable targets, such as URLs and the text adjacent to checkboxes, but our original implementation only targeted individual glyphs of text. Finally, in the seventh task (7) minor targets are tagged based on annotation. These tasks were designed to follow the natural progression of an implementation, simulating how a developer might build an enhancement in the wild.

Successes

When using our toolkit, participants took an average of 43 minutes to complete all tasks. No participants completed the baseline. Figure 7.9 presents a breakdown of completion times for each task. Most importantly, our toolkit enabled participants to implement core logic for a state-of-the-art enhancement and address unexplored shortcomings of that enhancement all within a relatively short study session.

To better understand of the difference in completion times, we conducted a semi-structured interview at the end of the study and examined the code authored by participants. We identified important differences between the conditions pertaining to code reuse and organization, data management, and the relationship between code and data.

Code Reuse and Organization

Participants wrote less code and reused a higher percentage of their code when using our toolkit. Specifically, they wrote an average of 36 lines using our toolkit versus 75 in the baseline. This difference was mostly due to our decomposition of code into layer chains. With our toolkit, participants would often reuse a layer several times in their chain, each parameterized with a different annotation library. In contrast, most of the code in the baseline was a result of participants copying from another enhancement, then editing that code to fit their enhancement. Participants reflected on these differences during the interview: P4 stated that *"the modular approach is good for code reusability",* and P3 said our toolkit *"is the great Unix way - piping inputs and outputs, reusing small methods"*.

Participants also noted that our toolkit yielded more clear and understandable solutions. For example, when asked to compare the two conditions, P4 mentioned that *"the pipeline was easier to reason about"* with our toolkit, and P6 stated that *"my organization was bad, mainly because I was copy-pasting stuff around"* in the baseline condition.

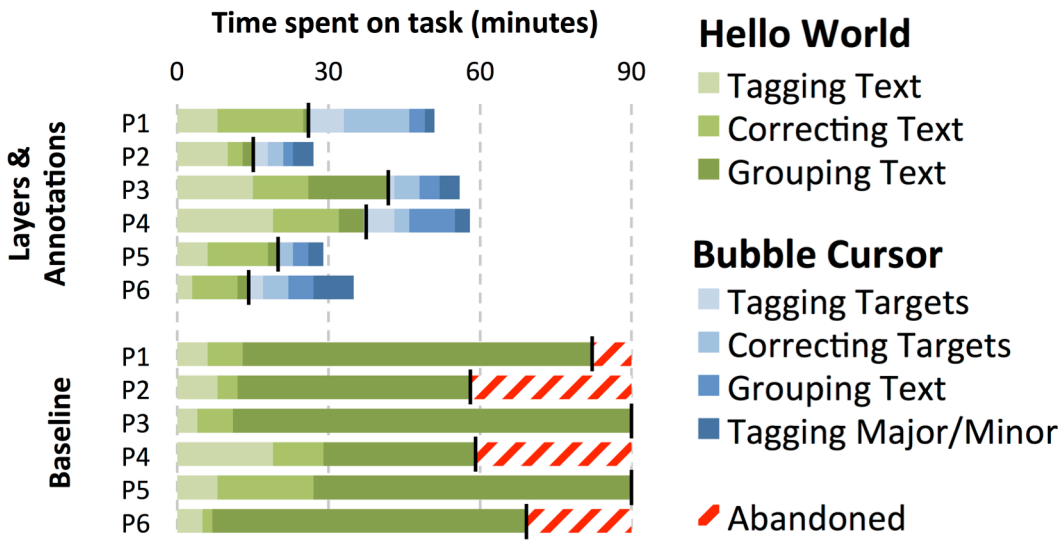


Figure 7.9: This graph presents each participant's completion times for each task. None of the participants were able to complete the baseline condition.

Participants also viewed our framework as more extensible. We asked them to explain which tools they would use for developing enhancements that are more sophisticated than the *Bubble Cursor*. P2 responded, *"I definitely would use [Prefab Layers and Prefab Annotations] because it would be easy to package up layers and give them to someone, and when I add new layers they don't break my code."* P1 stated, *"it would be easy to extend my code with some machine learning, because of the two-step process for importing and then using annotations."*

These successes demonstrate the value of decomposing code into layers. Participants were not required to write large amounts of code, but our toolkit still created a path of least resistance towards clear, extensible, and reusable code.

Understanding Annotations

Participants found our toolkit made it easy to understand the contents of their annotation libraries. In comparison, it was challenging to understand the path descriptors used in the baseline condition. This is partially because our toolkit provides the entire hierarchy containing an annotated element. Thus when debugging their code, participants were able to print the annotation node to the console, inspect its properties, and view its location in the image. In contrast, the path descriptors were cryptic because they only revealed the minimal details of an element's path needed to provide a unique reference. P6 mentioned this problem in the interview, stating, "*I liked the tree view a lot, as a way of navigating and examining elements*". This suggests Prefab Annotations are a better *expressive match* [55].

Relationship between Code and Data

Participants using our toolkit worked through each task as Emily did in our scenario walkthrough. In contrast, participants faced a critical roadblock in the baseline condition. This was to address subtle dependencies between annotations and code. This was exemplified in P2's session when implementing *Hello World*. The next paragraph steps through this as a comparison to our scenario walkthrough.

P2 successfully implemented functionality for task (1) that tagged leaf nodes as text. P2 then wrote code to correct erroneous tags using annotations and added a few corrections using the debugging tool. With the annotations working, P2 imported functionality for grouping text and added a few more corrections. However, these annotations did not cause any change in the output hierarchy. Confused, P2 spent several minutes inspecting code while adding and removing annotations. P2 then discovered the problem: they were storing path descriptors computed from the final hierarchy, which was different from the hierarchy in which corrections were applied before grouping. P2 thus rearranged their code such that the text corrections were applied after the grouping. However, this did not work either because the grouping code relied on those corrections to robustly determine which elements were text. At this point, P2 realized they would need to restructure the path descriptors to be consistent with the tree as it exists before grouping. P2 spent the remainder of their time developing this migration. Prefab Annotations removes the need for developers to migrate annotations between representations.

Challenges for Future Work

We also discovered challenges our participants faced when using our toolkit. Unanimously, participants found our toolkit to have a steeper learning curve than the baseline. Most of this difficulty was regarding the method for importing annotations. Specifically, participants initially did not see the benefit of writing explicit code to generalize annotations. This suggests that it might be useful for layers to generalize annotations by default, then allow developers to override the default with their own logic. Importantly, developers understood the point of defining their own generalizations, and reported that this added flexibility to implement more sophisticated enhancements.

Participants also noted that modularity introduced by layers would sometimes make it difficult to know when a piece of code would execute. Similarly, they wanted each layer to document its dependencies on any other layers, as they felt it was easy to lose track of how a single layer worked in the context of a larger chain. Ultimately, we see this overhead as a pervasive problem in modular programs and view this as a rich area for future work.

Participants in both conditions found it difficult to understand the computed path descriptors because they were based on the low-level pixel hierarchy recovered by Prefab. This raised a larger issue that elements in the hierarchy did not have human-readable names. Thus, it might be useful to include a default set of layers that tag Prefab's pixel-identified elements with such names.

Finally, participants requested the ability to more easily inspect the hierarchy at any point in a layer chain execution. This highlights a need for improvements in debugging tools, within which developers might toggle layers on and off or set breakpoints to view output at specific layers. We also believe there is an opportunity to explore IDEs for pixel-based methods, perhaps drawing on ideas from runtime modification or computer vision [43,46].

Prefab Layers and Prefab Annotations dramatically streamline the implementation of pixel-based runtime enhancements. Specifically, they help developers overcome subtle but critical dependencies between code and data. This toolkit is available at https://prefab.github.io, and we ultimately see this work as a step towards enabling the adoption of pixel-based methods in research and practice.

Chapter 8 | **CONCLUSION**

This dissertation aims to advance our understanding of how to design effective pixel-based methods for modifying interfaces at runtime. We therefore presented core methods for reverse engineering (Chapters 1 through 7), deeply explored enhancements enabled by these methods (Chapters 5 and 6), and introduced extensible developer tools for authoring new pixel-based methods (Chapter 7). Based on these explorations, this chapter identifies open challenges and opportunities for future work. We also generalize from our experiences and propose important guidelines for the design of future systems. Finally, we conclude with a summary of this research.

8.1 Training Example Collection

One core insight behind Prefab is that exact matching of raw pixel values can be used to rapidly identify interface elements. Example collection therefore becomes an important practicality of Prefab's approach. There exists a long tail of interface elements, each with a different appearance. Moreover, a single element typically has multiple appearances corresponding to its various states. While Prefab's separation of interface layout from appearance enables automatic prototype generation, it still requires the tedious collection of many examples to generate a sufficiently large prototype library. Future work might therefore explore methods for automatically collecting training examples from passive observation of interface usage. Alternatively, applications might incorporate lightweight mechanisms for collecting examples, similar to our Bubble Cursor's Target Editor and lightweight annotation menu presented in Chapter 4. Other research might investigate how online communities can collect and share libraries. Finally, we also believe it is worth exploring *unsupervised methods* for generating prototypes, removing the need to collect examples at all.

8.2 Hybrid Approaches to Reverse Engineering

Current pixel-based methods are fundamentally limited because they can only recover what is visible at the surface of the interface. In contrast, other researchers have explored leveraging deeper representations such as the DOM for a web page or the accessibility API. As mentioned in Chapter 2, these alternative representations are often incomplete or difficult to work with because of non-compliance from developers. To address these limitations, we believe it there is an opportunity to explore *hybrid* methods that automatically combine multiple representations of the same interface. Chang et al. presented initial explorations of hybrid methods in their PAX system, which used a combination of the accessibility API and Sikuli's pixel-based methods to more robustly recover textual content [11]. However, it may be

possible to develop methods that generalize beyond text recovery and beyond the single view of toolkits provided by the accessibility API. Specifically, the majority of interface toolkits represent UIs as hierarchies, so it could be possible to use *tree alignment algorithms* [41] to correlate nodes that represent the same interface element, and therefore bootstrap pixel-based hierarchies with important metadata gathered from many different views. More generally, these methods could enable developers to overcome the inherent incompleteness of any one view, while remaining independent of any particular toolkit or application's source code.

8.3 Supporting Input and Output Redirection

Most existing interface tools were not designed to support runtime modification, which creates several important challenges to implementing mechanisms for input and output redirection. One important limitation with current interface tools is that every application on a desktop shares the same input channel. For example, the global location of a single cursor is exposed to every application. Therefore, the fake target window in a Prefab runtime modification must precariously observe mouse movements without disrupting the underlying source window. For example, most of our applications work by 1) injecting logic into the global windowing system using a low-level mouse hook, 2) observing relative mouse movements and directing those to the target window, and 3) resetting the mouse position to a desired location over the underlying source window. Similar challenges occur in capturing the pixels of existing interfaces. Our methods use a low-level API call to force applications to render their pixels onto a buffer, which is obviously inefficient. Future work might simplify this process by *virtualizing input and output for each window*. For example, each application might expose its own *virtual cursor* that can be controlled independently from the global system cursor. Similarly, each application might output its pixel buffer to a virtual display that is exposed for runtime modification.

While the pixels of an interface can be interpreted independent of its underlying implementation, the challenges of robustly supporting input and output redirection points to the fact that *any runtime modification* will inevitably require some dependency on existing tools. Pixel-based methods minimize such dependencies and therefore lower the burden on developers for supporting new enhancements. Future work might also examine how pixel-based methods could be combined with entirely new toolkits developed from the ground up to further minimize such dependencies. For example, several modern toolkits are exploring functional approaches to programming interfaces, where *interface state* is explicitly separated from the specification of its appearance [56,61]. Such tools might therefore expose these state objects for deep runtime modification. Future pixel-based methods might then deduce the mapping between an application's appearance and its state object to support even more advanced behaviors.

8.4 Beyond Runtime Modification

This dissertation focuses on the use of pixel-based methods for the purposes of runtime modification, but these methods could also be used to implement a variety of applications that do not directly modify existing interfaces. Many examples of such applications are mentioned as related work in Chapter 2, but there are many other applications that have not yet been explored. For example, we are interested in investigating new methods for analyzing a long history of screenshots passively captured from a user's devices. These histories might then be intelligently summarized to provide the user with overviews and reminders of their tasks performed previously. They might also be used to automatically generate tutorials, or to passively identify common usability issues in existing applications.

8.5 Beyond the Desktop

We have primarily explored how to unlock interaction on the desktop, but we believe there is an enormous opportunity for unlocking interaction across many devices. For example, we envision improved speech interfaces on mobile devices that interpret existing applications and automate tasks on the user's behalf. It may also be possible to *automatically redesign interfaces to run on different devices*. For example, Chapter 6's discussion of Sliding Widgets mentions expanding beyond our initial methods for replacing individual widgets using a system like Supple [27]. Supple currently uses a manually crafted interface model to automatically generate an interface layout based on a particular user's abilities and a particular device's specifications. It may be possible to build upon our pixel-based methods to automatically deduce such a model from an existing interface, allowing developers to manually build an interface for one device, and then automatically generating designs optimized for many other contexts.

8.6 Beyond Developers

Today, most interfaces are designed by teams of people who are collocated and highly skilled. Moreover, any changes to an interface are implemented by the original developers and designers who own the source code. In contrast, we envision a future where distributed online communities rapidly construct and improve interfaces. Similar to the Wikipedia editing process, we imagine there could be new interface design tools that democratize the design of interfaces. These *community-based design tools* might include interfaces for directly manipulating existing interfaces in place, for voting on the best improvements to an existing design, educating new community members on important design principles, and sharing interface elements across applications.

8.7 Conclusion

Flexible interface tools are crucial to the progress of human-computer interaction. They shape what applications are possible and who can build them. Today, most interfaces are designed by teams of people who are collocated and highly skilled. Moreover, any changes to an interface are implemented by the original developers and designers who own the source code. In contrast, we envision a future where distributed online communities rapidly construct and improve interfaces. Similar to the Wikipedia editing process, we imagine new design tools that democratize the interface design process.

This dissertation contributes important initial steps towards unlocking existing interfaces from pixels. Specifically, we introduce novel pixel-based methods for reverse engineering graphical interfaces, which enable a practical approach to adding advanced behaviors to new and existing applications without access to their source code and independent of their underlying implementation. My work explores approaches to rapidly identifying interface elements from pixels by abstracting widget layout from widget appearance, methods for recovering interface content and hierarchy, an implementation and examination of a general-purpose enhancement, methods for recovering widget state and style, and developer support for composing and reusing pixel-based methods in arbitrary enhancements. This work also explores promising ideas proposed in the human-computer interaction literature that were previously impossible to implement in the context of real interfaces. This work therefore contributes important progress toward real-world deployment of an important family of research and sheds light on the gap between understanding research ideas in controlled settings versus real-world interfaces.

REFERENCES

1. Ahlström, D., Hitz, M., and Leitner, G. An Evaluation of Sticky and Force Enhanced Targets in Multi-Target Situations. *Proceedings of the Nordic Conference on Human-Computer Interaction*, (2006), 14–18.

2. Albinsson, P.-A. and Zhai, S. High Precision Touch Screen Interaction. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2003), 105–112.

3. Banovic, N., Grossman, T., Matejka, J., and Fitzmaurice, G. Waken: Reverse Engineering Usage Information and Interface Structure from Software Videos. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2012), 83–92.

4. Baudisch, P. and Chu, G. Back-of-Device Interaction Allows creating very Small Touch Devices. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2009), 1923–1932.

5. Baudisch, P., Cutrell, E., Robbins, D., et al. Drag-and-Pop and Drag-and-Pick : Techniques for Accessing Remote Screen Content on Touch-and Pen-Operated Systems. *INTERACT*, (2003), 57-64.

6. Baudisch, P., Tan, D., Collomb, M., and Robbins, D. Phosphor : Explaining Transitions in the User Interface Using Afterglow Effects. *Proceedings of the ACM Symposium on User Interface Software and Technology*, (2006), 169–178.

7. Benko, H., Wilson, A.D., and Baudisch, P. Precise Selection Techniques for Multi-Touch Screens. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2006), 1263–1272.

8. Blanch, R., Futurs, I., Guiard, Y., and Beaudouin-lafon, M. Semantic Pointing : Improving Target Acquisition with Control-Display Ratio Adaptation. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2004), 519–526.

9. Bolin, M., Webber, M., Rha, P., Wilson, T., and Miller, R.C. Automation and Customization of Rendered Web Pages. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2005), 163–172.

10. Casiez, G., Vogel, D., Balakrishnan, R., and Cockburn, A. The Impact of Control-Display Gain on User Performance in Pointing Tasks. *Human-Computer Interaction 23*, 3 (2008), 215–250.

11. Chang, T.-H., Yeh, T., and Miller, R. Associating the Visual Representation of User Interfaces with Their Internal Structures and Metadata. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2011), 245–254.

12. Chang, T.-H., Yeh, T., and Miller, R.C. GUI Testing Using Computer Vision. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2010), 1535–1544.

13. Chapuis, O., Labrune, J.-B., and Pietriga, E. DynaSpot : Speed-Dependent Area Cursor. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2009), 1391–1400.

14. Chapuis, O. and Roussel, N. UIMarks : Quick Graphical Interaction with Specific Targets. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2010), 173–182.

15. Cockburn, A. and Firth, A. Improving the Acquisition of Small Targets. *Human-Computer Interaction*.

16. Dixon, M., Fogarty, J., and Wobbrock, J. A General-Purpose Target-Aware Pointing Enhancement Using Pixel-Level Analysis of Graphical Interfaces. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2012), 3167–3176.

17. Dixon, M. and Fogarty, J. Prefab : Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2010), 1525–1534.

18. Dixon, M., Laput, G., and Fogarty, J. Pixel-Based Methods for Widget State and Style in a Runtime Implementation of Sliding Widgets. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2014), To appear.

19. Dixon, M., Leventhal, D., and Fogarty, J. Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2011), 969–978.

20. Eagan, J.R., Beaudouin-Lafon, M., and Mackay, W.E. Cracking the Cocoa Nut: User Interface Programming at Runtime. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2011), 225–234.

21. Edwards, W.K., Hudson, S.E., Marinacci, J., Rodenstein, R., Rodriguez, T., and Smith, I. Systematic Output Modification in a 2D User Interface Toolkit. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (1997), 151–158.

22. Findlater, L., Jansen, A., Shinohara, K., et al. Enhanced Area Cursors : Reducing Fine Pointing Demands for People with Motor Impairments. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2010), 153–162.

23. Findlater, L., Moffatt, K., Mcgrenere, J., and Dawson, J. Ephemeral Adaptation : The Use of Gradual Onset to Improve Menu Selection Performance. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2009), 1655–1664.

24. Fujima, J., Lunzer, A., Hornbæk, K., and Tanaka, Y. Clip, Connect, Clone: Combining Application Elements to Build Custom Interfaces for Information Access. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2004), 175.

25. Gaeremynck, Y., Bergman, L.D., and Lau, T. MORE for Less : Model Recovery from Visual Interfaces for Multi-Device Application Design. *Proceedings of the International Conference on Intelligent User Interfaces*, (2003), 69–76.

26. Gajos, K. and Weld, D.S. SUPPLE : Automatically Generating User Interfaces. .

27. Gajos, K.Z., Wobbrock, J.O., and Weld, D.S. Automatically Generating User Interfaces Adapted to Users' Motor and Vision Capabilities. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2007), 231–240.

28. Grossman, T. and Balakrishnan, R. The Bubble Cursor : Enhancing Target Acquisition by Dynamic Resizing of the Cursor 's Activation Area. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2005), 281–290.

29. Grossman, T., Matejka, J., and Fitzmaurice, G. Chronicle: Capture, Exploration, and Playback of Document Workflow Histories. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2010), 143–152.

30. Guiard, Y., Blanch, R., and Beaudouin-lafon, M. Object Pointing : A Complement to Bitmap Pointing in GUIs. *GI*, (2004), 9–16.

31. Hartmann, B., Wu, L., Collins, K., and Klemmer, S.R. Programming by a Sample: Rapidly Creating Web Applications with d.mix. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2007), 241–250.

32. Hartmann, B., Yu, L., Allison, A., Yang, Y., and Klemmer, S.R. Design as Exploration : Creating Interface Alternatives through Parallel Authoring and Runtime Tuning. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2008), 91–100.

33. Hourcade, J.P., Perry, K.B., and Sharma, A. PointAssist : Helping Four Year Olds Point with Ease. *IDC*, (2008), 202–209.

34. Hudson, S.E. and Smith, I. Supporting Dynamic Downloadable Appearances in an Extensible User Interface toolkit. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (1997), 159–168.

35. Hudson, S.E. and Tanaka, K. Providing Visually Rich Resizable Images for User Interface Components. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2000), 227–235.

36. Hurst, A., Hudson, S.E., and Mankoff, J. Automatically Identifying Targets Users Interact with During Real World Tasks. *Proceedings of the International Conference on Intelligent User Interfaces*, ACM Press (2010), 11–20.

37. Hurst, A., Mankoff, J., Dey, A.K., and Hudson, S.E. Dirty Desktops : Using a Patina of Magnetic Mouse Dust to Make Common Interactor Targets Easier to Select. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2007), 183–186.

38. Hutchings, D.R. and Stasko, J. mudibo : Multiple Dialog Boxes for Multiple Monitors. *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems*, (2005), 1471–1474.

39. Hwang, F., Keates, S., Langdon, P., and Clarkson, P.J. Multiple Haptic Targets for Motion-Impaired Ccomputer Users. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2003), 41–48.

40. Jansen, A., Findlater, L., and Wobbrock, J.O. From the Lab to the World : Lessons from Extending a Pointing Technique for Real-World Use. *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems*, (2011), 1867–1872.

41. Jiang, T., Wang, L., and Zhang, K. Alignment of Trees - An Alternative to Tree Edit. *Theoretical Computer Science 143*, (1995), 137–148.

42. Kabbash, P. and Buxton, W. The " Prince " Technique : Fitts ' Law and Selection Using Area Cursors. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (1995), 273–279.

43. Kato, J., Mcdirmid, S., and Cao, X. DejaVu : Integrated Support for Developing Interactive Camera-Based Programs. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2012), 189–196.

44. Kelleher, C. and Pausch, R. Stencils-Based Tutorials : Design and Evaluation. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2005), 541–550.

45. Kushmerick, N., Weld, D.S., and Doorenbos, R. Wrapper Induction for Information Extraction. 1997, Proceedings of the International Joint Conferences.

46. Meng, X., Zhao, S., Huang, Y., Zhang, Z., and Eagan, J.R. WADE : Simplified GUI Add-on Development for Third-party Software. (2014), 2221–2230.

47. Moscovich, T. Contact Area Interaction with Sliding Widgets. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2009), 13–22.

48. Myers, B., Hudson, S.E., and Pausch, R. Past, Present, and Future of User Interface Software Tools. *Transactions on Computer-Human Interaction 7*, 1, 3–28.

49. Mynatt, D. and Edwards, W.K. Mapping GUIS to Auditory Interfaces. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (1992), 61–70.

50. Nichols, J., Hua, Z., and Barton, J. Highlight: A System for Creating and Deploying Mobile Web Applications. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2008), 249–258.

51. Nichols, J. and Lau, T. Mobilization by Demonstration: Using Traces to Re-author Existing Web Sites. *Proceedings of the International Conference on Intelligent User Interfaces*, ACM Press (2008), 149–160.

52. Nichols, J., Myers, B. a., and Litwack, K. Improving Automatic Interface Generation with Smart Templates. *Proceedings of the International Conference on Intelligent User Interfaces*, ACM Press (2004), 286–288.

53. Olsen, D.R., Hudson, S.E., Verratti, T., Heiner, J.M., and Phelps, M. Implementing Interface Attachments Based on Surface Representations. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (1999), 191–198.

54. Olsen, D.R., Taufer, T., and Fails, J.A. ScreenCrayons: Annotating Anything. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2004), 165–174.

55. Olsen, D.R. Evaluating User Interface Systems Research. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2007), 251–258.

56. Oney, S., Myers, B., and Brandt, J. InterState : A Language and Environment for Expressing Interface Behavior. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2014), 263–272.

57. Pongnumkul, S., Dontcheva, M., Li, W., et al. Pause-and-Play: Automatically Linking Screencast Video Tutorials with Applications. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2011), 135–144.

58. Potter, R. Triggers: Guiding Automation with Pixels to Achieve Data Access. *A. Cypher, eds. MIT Press*.

59. Rissanen, J. Modeling by Shortest Data Description. *Automatica 14*, 5 (1978), 465–471.

60. Russel, S. and Norvig, P. *Artificial Intelligence: A Modern Approach.* 2002.

61. Satyanarayan, A., Wongsuphasawat, K., and Heer, J. Declarative Interaction Design for Data Visualization. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2014), 669–678.

62. St Amant, R., Lieberman, H., and Potter, R. Visual Generalization in Programming by Example. *Communications of the ACM 43*, 3 (2000), 107–114.

63. St Amant, R., Riedl, R., Ritter, F.E., and Reifers, A. Image Processing in Cognitive Models with SegMan. *HCII*, (2005).

64. Stuerzlinger, W., Chapuis, O., Phillips, D., and Roussel, N. User Interface Façades: Towards Fully Adaptable User Interfaces. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2006), 309–318.

65. Takagi, H. Social Accessibility : Achieving Accessibility through Collaborative Metadata Authoring. *Proceedings of the ACM SIGACCESS Conference on Computers and Accessibility (Assets)*, ACM Press (2008), 193–200.

66. Tan, D.S., Meyers, B., and Czerwinski, M. WinCuts : Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2004), 1525–1528.

67. Terry, M. and Mynatt, E.D. Side Views : Persistent , On-Demand Previews for Open-Ended Tasks. *Proceedings of the ACM Symposium on User Interface Software and Technology*, (2002), 71–80.

68. Vogel, D. and Baudisch, P. Shift : A Technique for Operating Pen-Based Interfaces Using Touch. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2007), 657–666.

69. Waldner, M., Steinberger, M., Grasset, R., and Schmalstieg, D. Importance-Driven Compositing Window Management: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2011), 959–968.

70. Weldon, J. and Shneidermanss, B. Improving the Accuracy of Touch Screens: An Experimental Evaluation of Three Strategies. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (1988), 27–32.

71. Wigdor, D., Forlines, C., Baudisch, P., Barnwell, J., and Shen, C. LucidTouch : A See-Through Mobile Device. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2007), 269–278.

72. Wobbrock, J.O. The Angle Mouse : Target-Agnostic Dynamic Gain Adjustment Based on Angular Deviation. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (2009), 1401–1410.

73. Worden, A., Walker, N., Bharat, K., and Hudson, S.E. Making Computers Easier for Older Adults to Use : Area Cursors and Sticky Icons. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (1997).

74. Yeh, T., Chang, T.-H., and Miller, R.C. Sikuli: Using GUI Screenshots for Search and Automation. *UIST*, ACM Press (2009), 183–194.

75. Yeh, T., Chang, T.-H., Xie, B., et al. Creating Contextual Help for GUIs Using Screenshots. *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press (2011), 145–154.

76. Zettlemoyer, L.S. and St. Amant, R. A Visual Medium for Programmatic Control of Interactive Applications. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press (1999), 199–206.

77. Zettlemoyer, L.S., Amant, R.S., and Dulberg, M.S. IBOTS : Agent Control Through the User Interface. *Proceedings of the International Conference on Intelligent User Interfaces*, (1999).