Prefab Layers and Prefab Annotations: Extensible Pixel-Based Interpretation of Graphical Interfaces

Morgan Dixon, Conrad Nied, and James Fogarty
Computer Science & Engineering
DUB Group, University of Washington
{mdixon,anied,jfogarty}@cs.washington.edu

ABSTRACT
Pixel-based methods have the potential to fundamentally change how we build graphical interfaces, but remain difficult to implement. We introduce a new 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. Together, these 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.

Author Keywords
Prefab; layers; annotations; pixel-based reverse engineering.

ACM Classification Keywords
H.5.m. Information interfaces and presentation (e.g., HCI).

INTRODUCTION AND MOTIVATION
Pixel-based methods can enable modification of interfaces without their code, independent of their implementation. For example, the applications in Figure 1 use pixel-based methods to enhance the entire desktop. The first is an implementation of the Bubble Cursor [5,12], which dynamically resizes to always select the nearest target. The next is an implementation of sliding widgets [7,18], which replaces mouse-based interface elements with touchscreen widgets. The third translates the language of interfaces for improved localization [8]. Other examples include accessibility enhancements, testing frameworks, automation tools, and help systems [6,8,30,31]. These enhancements modify interfaces in a variety ways, but they are all enabled by methods that use pixels as a universal representation.

UIST '14, October 05 - 08 2014, Honolulu, HI, USA.

http://dx.doi.org/10.1145/2642918.2647412

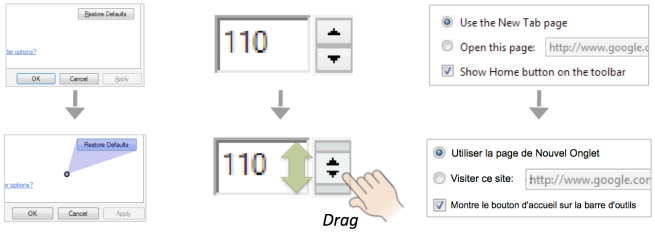


Figure 1: We present a toolkit that streamlines the implementation of pixel-based enhancements. Three examples supported by our toolkit include the Bubble Cursor [5,12], sliding widgets [7,18], and interface language translation [8].

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 [5]. 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 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 work 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 interface 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.

RELATED WORK

We focus on pixel-based interpretation of graphical interfaces, as applied to runtime modification of existing interfaces. This section overviews prior runtime modification work, and then looks more specifically at the current strengths and limitations of pixel-based methods.

Runtime Modification

Runtime modification has broad applications in accelerating innovation and facilitating adoption. In classic work, Edwards et al. [10] and Olsen et al. [22] 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. More recent examples of runtime modification leverage the open nature of a website's Document Object Model (DOM) to access and modify existing interfaces. Web-based modifications include creating mash-ups between existing applications [11,14,28], re-authoring web applications for mobile interfaces [20,21], and automating repetitious interactions [2,28].

Although websites expose a DOM, traditional approaches to modifying desktop applications are based in accessibility APIs [28] or injecting into an interface toolkit [9,10,22]. Accessibility APIs expose interface state, but unfortunately are frequently incomplete because application developers fail to implement the API. For example, Hurst et al. found 25% of elements are completely missing [15]. Injection techniques insert custom logic into an interface via the toolkit or other runtime system. However, injection must be carefully crafted for each interface and underlying toolkit. This limits the utility of injection for general-purpose enhancements, as people typically use a variety of applications implemented with several toolkits.

In contrast, pixel-based methods do not require cooperation from the developers of an interface and circumvent fragmentation of interfaces and toolkits. Leveraging these advantages, researchers have explored a variety of pixel-based approaches to recover and modify the structure of an interface. St. Amant's Segman uses hand-crafted code to identify specific types of interface elements [27]. Sikuli uses template matching and voting on local features for image-based identification of interface elements [30]. Savva uses more sophisticated computer vision techniques for automated visualization retargeting [26]. Finally, PAX presents a hybrid approach, supplementing the accessibility API with pixel-based processing [3].

Flexibility and Robustness of Pixel-Based Methods

Prior pixel-based methods enable a variety of modifications to existing interfaces, including contextual and video-based tutorials [1,25,31], interface testing frameworks [4], new window managers [29], note-taking overlays [23], and systems for exploring document workflow histories [13]. Although these begin to demonstrate the potential of pixel-based methods, most of this initial success is based on a low-level understanding of existing interfaces (e.g., the locations of salient regions in an image, the position of a single element that matches a specific template). There are few enhancements based on higher-level interpretations, as current tools are not flexible or robust enough to support multiple levels of interpretation.

Althought it is difficult to recover high-level information from pixels, the Bubble Cursor and sliding widgets enhancements in Figure 1 require relatively sophisticated interpretation. Their success is due to their combination of interpretation code and human annotation of identified elements. Specifically, their systems heuristically infer semantic information about interface elements and then use human-provided corrections to override erroneous inferences. The goal of our toolkit is to support these and other complex enhancements. Our validation demonstrates that developers can build the same methods used by the state-of-the-art Bubble Cursor implementation, and we also show how our methods allow developers to easily extend the cursor to support more advanced behaviors. We describe the details of other enhancements enabled by our toolkit, and we also describe how annotations and code can be shared among multiple enhancements.

We build on Prefab's methods for reverse engineering interface structure [6,8]. Prefab identifies interface elements from pixels and organizes them into a hierarchy. The root corresponds to the processed image, and identified elements are added as children to elements in which they are spatially contained. This spatial hierarchy is not the same as an interface's logical hierarchy, but represents visible containment (e.g., buttons, group boxes, tab panes).

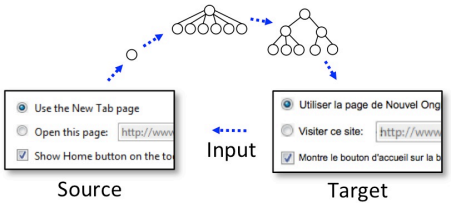


Figure 2: We present a new toolkit that structures pixel-based interpretation as a series of tree transformations.

Importantly, this tree only provides identified elements and does not include metadata about elements (e.g., whether they are targets, their widget type). The following sections describe how developers use our methods to build high-level interpretations on Prefab's low-level hierarchy.

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 1.

Toolkit Overview

As in prior work [5,6,7,8], our methods are designed to be combined with input and output redirection. Figure 2 illustrates: (1) a source window bitmap is captured, (2) the source image is interpreted, (3) a modified interface is presented in a target window, (4) input in the target window is mapped back to the source, which then (5) generates new output that is captured to update the target. Runtime modification is realized by rapidly and repeatedly executing this cycle. Implementing this cycle is hard because of the lack of support for interpreting the source image in step (2).

To address this challenge, our toolkit decomposes interpretation into a series of tree transformations, as shown at the top of Figure 2. 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.

Our 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 paper 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 1. Emily has found many applications do not support her native 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 3 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 2, 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.

```
# main.py
import prefab_layers
chain = prefab_layers.new_chain()
chain.import_layer('prefab_identification')
chain.import_layer('text_recovery')
```

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.

```
# 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)
```

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.

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

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

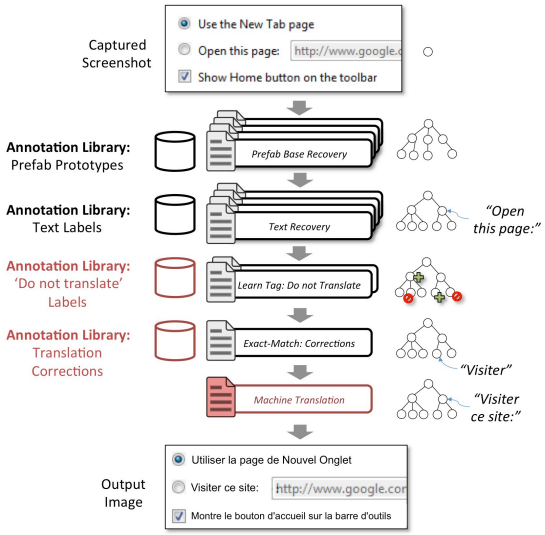


Figure 3: Our toolkit simplifies runtime interface translation. It uses layers that recover interface text, decide what text should be translated, and then present translations obtained using both machine translation and human correction.

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.

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

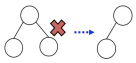
Prefab Layers therefore gives each layer an *immutable* view on its input. We provide a set of tree transformation operations, and layers can request any number of operations be applied to nodes in the hierarchy. All operations are then applied in batch after the layer terminates (i.e., the hierarchy 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 a hierarchy. 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. [19], 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.

Our specific operations are *tagging* a node, *setting an ancestor* for a node, and *deleting* a node. We chose these operations 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 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).

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). 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,

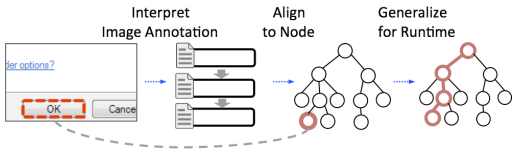


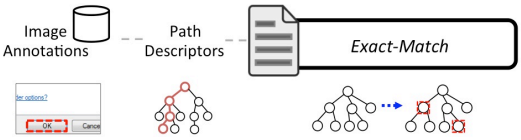
Figure 4: When a layer imports an annotation library, we provide a tree representation for each annotated element. Image annotations are interpreted by preceding layers, aligned to nodes in the resulting hierarchies, and then handed to the layer, where it computes information it will need at runtime.

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 4 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 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 5 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.




```
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 5: 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.

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 [24], 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 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 paper, 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 [8]. Other work explores incorporating text from the accessibility API [3]. 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.

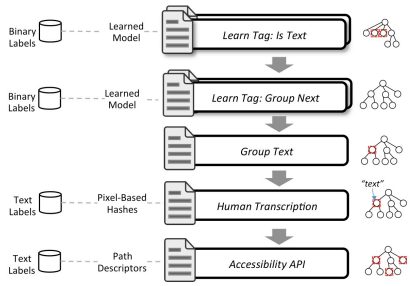


Figure 6: Layer chains can implement sophisticated reusable interpretation logic. Our text recovery chain combines human transcription with text recovered from the accessibility API.

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 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 [8], 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 2 and our introductory scenario present a new and more extensible implementation addressing these issues. Our 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

Our prior Bubble Cursor implementation is an example of the state-of-the-art in pixel-based enhancements [5,12]. It enables target-aware pointing across the entire desktop, identifying targets using Prefab and 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.

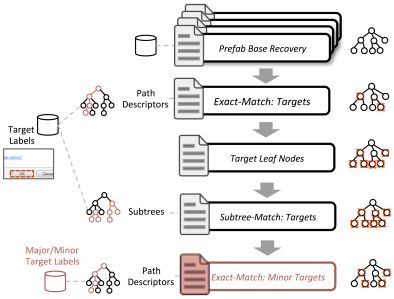


Figure 7: We streamline and expand our Bubble Cursor enhancement [5], resolving important limitations of our previous implementation.

Our new toolkit dramatically streamlines this enhancement. Figure 7 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 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 two previous layer chains highlight re-implementations of prior systems, but we have also used our toolkit to implement an entirely new enhancement. As shown in Figure 1, sliding widgets are touchscreen widgets activated by sliding a moveable element [7,18]. In the course of developing our toolkit, we implemented an enhancement that replaces mouse-based widgets with sliding widgets. A detailed presentation is presented in our recent work [7].

The sliding widgets enhancement would have been tedious or impractical to develop without our 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. Our toolkit also provides a clear conceptual model for reasoning about and describing our sliding widgets enhancement. Using figures similar to the layer chain illustrations throughout this paper, that paper is able to succinctly overview the code and data used in the enhancement [7]. Our current contribution is therefore not

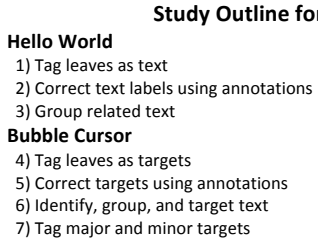


Figure 8: Participants were given 7 tasks in each condition. The first three implement a *Hello World* enhancement. The next four implement and expand upon the *Bubble Cursor*.

only code, but also a set of abstractions that allow effective description of pixel-based interpretation.

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 [5]. 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 [30], 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. Our 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.

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 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 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.

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* [24].

Relationship between Code and Data

Participants using our toolkit worked through each task as Emily did in our scenario walkthrough. In contrast,

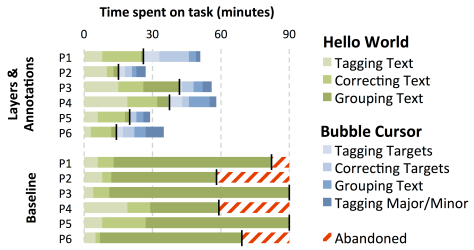


Figure 9: This graph presents each participant's completion times for each task. None of the participants were able to complete the baseline condition.

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 and Conclusion

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 [9,16,17].

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://github.com/prefab, and we ultimately see this work as a step towards enabling the adoption of pixel-based methods in research and practice.

ACKNOWLEDGMENTS

We thank Jeffrey Heer, Dan Weld, and Jacob Wobbrock for discussions related to this work. This work was supported by the National Science Foundation under award IIS-1053868.

REFERENCES

1. Banovic, N., Grossman, T., Matejka, J., and Fitzmaurice, G. Waken: Reverse Engineering Usage Information and Interface Structure from Software Videos. *UIST 2012*. 83–92.
2. Bolin, M., Webber, M., Rha, P., Wilson, T., and Miller, R.C. Automation and Customization of Rendered Web Pages. *UIST 2005*. 163–172.
3. Chang, T.-H., Yeh, T., and Miller, R. Associating the Visual Representation of User Interfaces with Their Internal Structures and Metadata. *UIST 2011*. 245–254.
4. Chang, T.-H., Yeh, T., and Miller, R.C. GUI Testing Using Computer Vision. *CHI 2010*. 1535–1544.
5. Dixon, M., Fogarty, J., and Wobbrock, J. A General-Purpose Target-Aware Pointing Enhancement Using Pixel-Level Analysis of Graphical Interfaces. *CHI 2012*. 3167–3176.
6. Dixon, M. and Fogarty, J. Prefab : Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. *CHI 2010*. 1525–1534.
7. Dixon, M., Laput, G., and Fogarty, J. Pixel-Based Methods for Widget State and Style in a Runtime Implementation of Sliding Widgets. *CHI 2014*. 2231-2240.
8. Dixon, M., Leventhal, D., and Fogarty, J. Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure. *CHI 2011*. 969–978.
9. Eagan, J.R., Beaudouin-Lafon, M., and Mackay, W.E. Cracking the Cocoa Nut: User Interface Programming at Runtime. *UIST 2011*. 225–234.
10. Edwards, W.K., Hudson, S.E., Marinacci, J., Rodenstein, R., Rodriguez, T., and Smith, I. Systematic Output Modification in a 2D User Interface Toolkit. *UIST 1997*. 151–158.
11. Fujima, J., Lunzer, A., Hornbæk, K., and Tanaka, Y. Clip, Connect, Clone: Combining Application Elements to Build Custom Interfaces for Information Access. *UIST 2004*. 175-184.
12. Grossman, T. and Balakrishnan, R. The Bubble Cursor : Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area. *CHI 2005*. 281–290.
13. Grossman, T., Matejka, J., and Fitzmaurice, G. Chronicle: Capture, Exploration, and Playback of Document Workflow Histories. *UIST 2010*. 143–152.
14. Hartmann, B., Wu, L., Collins, K., and Klemmer, S.R. Programming by a Sample: Rapidly Creating Web Applications with d.mix. *UIST 2007*. 241–250.
15. Hurst, A., Hudson, S.E., and Mankoff, J. Automatically Identifying Targets Users Interact with During Real World Tasks. *IUI 2010*. 11–20.
16. Kato, J., Mcdirmid, S., and Cao, X. DejaVu : Integrated Support for Developing Interactive Camera-Based Programs. *UIST 2012*. 189–196.
17. Meng, X., Zhao, S., Huang, Y., Zhang, Z., and Eagan, J.R. WADE : Simplified GUI Add-on Development for Third-party Software. *CHI 2014*. 2221–2230.
18. Moscovich, T. Contact Area Interaction with Sliding Widgets. *UIST 2009*. 13–22.
19. Myers, B., Hudson, S.E., and Pausch, R. Past, Present, and Future of User Interface Software Tools. *TOCHI 7 (1)*. 3–28.
20. Nichols, J., Hua, Z., and Barton, J. Highlight: A System for Creating and Deploying Mobile Web Applications. *UIST 2008*. 249–258.
21. Nichols, J. and Lau, T. Mobilization by Demonstration: Using Traces to Re-author Existing Web Sites. *IUI 2008*. 149–160.
22. Olsen, D.R., Hudson, S.E., Verratti, T., Heiner, J.M., and Phelps, M. Implementing Interface Attachments Based on Surface Representations. *CHI 1999*. 191–198.
23. Olsen, D.R., Taufer, T., and Fails, J.A. ScreenCrayons: Annotating Anything. *UIST 2004*. 165–174.
24. Olsen, D.R. Evaluating User Interface Systems Research. *UIST 2007*. 251–258.
25. Pongnumkul, S., Dontcheva, M., Li, W., Wang, J., Bourdev, L., Avidan, S., and Cohen, M. Pause-and-Play: Automatically Linking Screencast Video Tutorials with Applications. *UIST 2011*. 135–144.
26. Savva, M., Kong, N., Chhajta, A., Fei-fei, L., Agrawala, M., and Heer, J. ReVision: Automated Classification, Analysis and Redesign of Chart Images. *UIST 2011*. 393–402.
27. St Amant, R., Riedl, R., Ritter, F.E., and Reifers, A. Image Processing in Cognitive Models with SegMan. *HCII 2005*.
28. Stuerzlinger, W., Chapuis, O., Phillips, D., and Roussel, N. User Interface Façades: Towards Fully Adaptable User Interfaces. *UIST 2006*. 309–318.
29. Waldner, M., Steinberger, M., Grasset, R., and Schmalstieg, D. Importance-Driven Compositing Window Management: CHI 2011. 959–968.
30. Yeh, T., Chang, T.-H., and Miller, R.C. Sikuli: Using GUI Screenshots for Search and Automation. *UIST 2009*. 183–194.
31. Yeh, T., Chang, T.-H., Xie, B.,Walsh, G., Watkins, I., Wongsuphasawat, K., Huan, M., Davis, L.S., Bederson, B. Creating Contextual Help for GUIs Using Screenshots. *UIST 2011*. 145–154.