Demo: UIWear: Easily Adapting User Interfaces for Wearable Devices

Jian Xu*
Stony Brook University
jianxu1@cs.stonybrook.edu

Qingqing Cao*
Stony Brook University
qicao@cs.stonybrook.edu

Aruna Balasubramanian
Stony Brook University
arunab@cs.stonybrook.edu

Donald E. Porter
The University of North Carolina at Chapel Hill
porter@cs.unc.edu

ABSTRACT

Wearable devices, such as smart watches, offer exciting new opportunities for users to interact with their applications. The current state of the art for wearable devices is for a developer to write a custom *companion app*, which is a variant of the smartphone app, tailored to the wearable form factor. A developer puts a non-trivial amount of effort to write these companion apps and the programming model does not scale to an increasing diversity of form factors.

In this demo, we show a working prototype of our system UIWear that allows a developer to easily extend a smartphone application to other wearable interfaces. Our system, UIWear, extracts the application GUI as a UI tree, which preserves the semantics of the GUI. The developer (or the user) only writes a *metaprogram* to encode the GUI design for the wearable device; no effort is needed beyond the design phase. UIWear executes the metaprogram by performing all the underlying tasks to virtualize the application GUI, adapt it, and recreate it on the wearable. A metaprogram can create the same functionality as existing companion apps with an order-of-magnitude less programming effort.

1 INTRODUCTION

Wearable computing devices are a major growth sector for computing, and represent a step torward practical, ubiquitous computing. Wearable devices decouple the benefits of computer assistance from sitting at a computer or even holding a bulky phone. Wearable apps have applications in diverse areas, including healthcare, personal assistance, navigation, personal security, and many more. For example, GPS heads-up display helps skiers navigate a difficult slope [4]. Similarly, the *Spotify* smartwatch app allows the user to control the music easily, freeing the user's hands from repeatedly taking out the phone.

A prevailing programming model for wearable devices is to create *companion apps* that act as a companion for apps on the more powerful mobile phone. A companion app extends a subset of the smartphone app's graphical user interface (GUI) to the wearable device, adapting the GUI for a different form factor. For example, the *Spotify* companion app (see Figure 1) on the user's watch exports a view of the *Spotify* smartphone app to the watch, and events such as pausing the song are synchronized with the smartphone app.

Unfortunately, the programming model for wearable devices simply does not scale. For example, the *Spotify* app needs to be extended to the smartglass, a different companion app is required. These companion apps can be over a thousand lines of code (and tens of thousands of lines of code including libraries), and require the original app developer to manage the UI, emulate user interaction, and write an ad hoc RPC protocol to synchronize the wearable and the phone application [11]. Because of the significant developer effort required to create a companion app, only a small fraction of smartphone apps have companion apps. Even popular apps such as Facebook do not have a companion app. Worse yet, without source code, third-party developers cannot design new companion apps for an existing smartphone app nor adjust a companion app to better suit their needs.

In this demo, we present a different programming model for wearable devices, where the developer writes the smartphone application *once*, and writes a simple *meta program* to specify how the application should be extended to a given form factor. Our system, UIWear, automatically creates companion apps, that we call *UICompanion* apps, for different form factors based on the meta program. The developer effort in writing the meta program is an order-of-magnitude lower than writing the companion app. UIWear is transparent to the application, does not require source code or changes to the original application, can re-tailor the GUI to suit the wearable interface, and is well-suited for mobile and wearables.

The overarching principle of UIWear is to abstract application management from application design. The human designer (developer or end-user) only specifies what parts of the smartphone GUI are mapped to the wearable device and how the GUI is re-tailored, using a simple *metaprogram*. Writing a metaprogram requires substantially lower programming effort than writing a full-fledged companion app, as the developer needs only specify the interface design.

UIWear's *UI virtualization architecture* manages all other aspects of extending the app to the wearable device. UIWear abstracts a logical model of the GUI as a loose set of UI elements, and uses this abstraction to re-tailor and render the GUI on the wearable. UIWear also performs all synchronization at the GUI layer using the logical GUI model. All user interactions on the wearable device are

*Primary authors with equal contribution.

MobiCom '17, October 16–20, 2017, Snowbird, UT, USA

ACM ISBN 978-1-4503-4916-1/17/10.
https://doi.org/10.1145/3117811.3124769

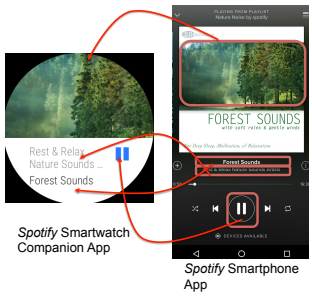


Figure 1: The *Spotify* smartphone and the corresponding smartwatch companion app. A subset of the smartphone GUI (marked within the *red* box) is re-tailored for the watch.

captured by the GUI abstraction and sent to the phone. The phone emulates the user interaction as though it was a local event, and the resulting update to the phone GUI is mirrored on the wearable.

One essential contribution of UIWear is identifying and extracting a logical model of the GUI. UIWear uses the *UI tree* abstraction, which is commonly used across operating systems to represent the GUI as individual UI elements and their relationships [1, 2]. UIWear essentially splices the UI tree, tailors it according to the metaprogram, and compiles it into the UICompanion app. A key contribution of UIWear is in extracting the UI tree from the applications transparently.

Another key challenge is in synchronizing at the GUI layer. Smartphones only allow a *single* active GUI at a given time; in fact, the graphics stack of background apps is destroyed. The consequence is that the wearable app can only synchronize as long as the phone app is in the foreground. We observe that, while maintaining the full graphics stack of each background application is power consuming, the logical GUI model (in our case the UI tree) can be kept active even for background apps with minimal effect on power. In UIWear, we modify the operating system to keep the UI tree of a background application active. This simple change allows us to multiplex I/O events to and from background applications with minimal effect on power.

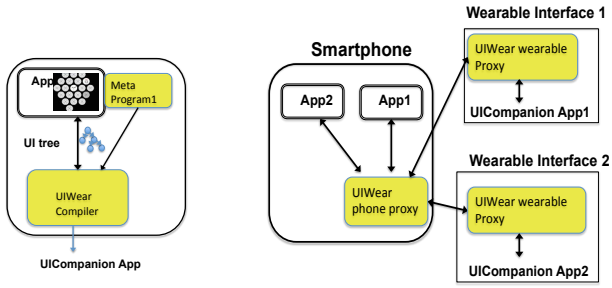
There has been related work both in the systems and the human-computer interaction (HCI) communities for designing cross-device applications. However, related systems either only work in homogenous environments that do not require UI re-tailoring [7, 8], or only work for specific applications and devices while supporting UI tailoring [9, 10, 12]. UIWear combines the advantages of the two approaches.

2 UIWEAR

Companion app Background. Companion apps are a common programming paradigm used by wearable operating systems and vendors including AndroidWear, Apple, Tizen, Pebble, and the now defunct Google glass. A companion app is a GUI projection of the original smartphone app, where parts of the smartphone UI are re-tailored and displayed on the wearable.

Figure 1 shows the *Spotify* companion app from the AndroidWear playstore [6] for smartwatches. The companion app exports a view of the Spotify smartphone app to the watch, and events such as pausing the song are synchronized with the smartphone.

UIWear Architecture. The UIWear architecture has two stages: compile time and runtime, as shown in Figures 2(a) and (b). At compile time, the human designer writes a metaprogram and UIWear compiles this to the UICompanion app. The metaprogram is the only step that requires a human-in-the-loop. The developer writes a separate metaprogram for each wearable interface such as watch and a glass. The metaprogram includes decisions about what parts of the UI are to appear on the wearable device and how the UI is re-tailored.



(a) UIWear Compiler and Metaprogram: UIWear compiles the metaprogram specification to create the UICompanion app.

(b) At UIWear Runtime: The UICompanion app on the wearable device synchronizes state with the smartphone app to relay user interactions and get UI updates.

Figure 2: UIWear Architecture

The UIWear compiler compiles the metaprogram to the UICompanion app and ships the app to the wearable device. UIWear extracts the UI tree to abstract and re-tailor the GUI. A natural point at which to extract the UI tree is at the accessibility and UI automation interface. Current accessibility interfaces are designed only for extracting textual content of the GUI, primarily for applications for the visually impaired that may not require graphical content. UIWear augments this existing technique so that the complete UI tree, including text and visual data are extracted.

At run time, UIWear synchronizes the UI states of the wearable app and the corresponding phone app. UIWear's synchronization protocol relays input events from the wearable device to the UIWear phone proxy, which emulates the user input on the phone. The resulting GUI update is relayed to the wearable device using the UI tree abstraction. This synchronization requires no support from the phone application, unlike existing companion apps that need to write a custom remote procedure call.

Implementation. We implement UIWear on the Android ecosystem: Android phone OS, AndroidWear smartwatch OS, and the Sony SmartEyeGlass that runs over a version of Android [5].

From the end-user's perspective, an Android application is made up of multiple "windows" or activities. The designer writes a metaprogram for each application window to be mapped to the wearable, and UIWear compiles the metaprograms and bundles them into a single UICompanion app. UIWear uses the standard Android toolchain to create the UICompanion application. The application consists of the source file, the AndroidManifest file, the layout, and

a set of resources that are created programmatically. We create templates for the metaprogram using FreeMarker [3].

We implement the UIWear phone proxy as a client to the accessibility service on Android, and the wearable proxy as a service on the wearable device. The phone and wearable proxies together contain 5,300 lines of code. The phone proxy keeps track of preference files and wearable UI tree for different applications as well as different windows of the same application. The UIWear watch and phone proxy exchange information over the Google Messaging Service (GMS). GMS encapsulates the data and performs the low-level communication tasks, and can use Bluetooth or WiFi.

To multiplex I/O between several UICompanion apps and background apps, UIWear modifies Android to allow the graphic stack to sleep, but keep the intermediate UI representation active.

3 DEMO DETAILS

In this demonstration, we will show how the UICompanion app is generated by UIWear based on the metaprogram specification. We will also show the corresponding developer-written companion apps downloaded from AndroidWear Marketplace. The audience can interact with both versions of the app and compare the look-and-feel and functionality between two versions. We will also show how UIWear can create wearable applications for smartphone apps for which no wearable application currently exists. Specifically, we will demonstrate the companion app creation for *Facebook messenger* and *Yahoo Mail*, as well as companion app creation on Smartglasses. Interested audience members will be invited to write their own metaprogram for extending smartphone apps to wearables.

The demonstrations will be performed on a Nexus 5 smartphone running a customized Android Marshmallow 6.0.1 OS, a smartwatch running AndroidWear 1.5, the Sony SmartEyeglass, and a laptop, all of which will be bought to the venue by the authors. The demonstration requires a desk, power outlets, and WiFi. The demo setup will take less than 30 minutes.

REFERENCES

[1] Android View Hierarchy. http://developer.android.com/guide/topics/ui/overview.html.

[2] Apple View Hierarchy. https://developer.apple.com/library/ios/documentation/General/Conceptual/Devpedia-CocoaApp/View%20Hierarchy.html.

[3] Freemarker. http://freemarker.org/.

[4] GPS Heads Up Display. http://www.reconinstruments.com/products/snow2/.

[5] Sony SmartEyeGlass SDK. https://developer.sony.com/develop/wearables/smarteyeglass-sdk/.

[6] Google Android Wear Market. https://play.google.com/store/apps/category/ANDROID_WEAR?hl=en.

[7] J. Andrus, A. Van't Hof, N. AlDuaij, C. Dall, N. Viennot, and J. Nieh. Cider: Native execution of ios apps on android. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 367–382, 2014.

[8] Apportable. http://www.apportable.com/.

[9] S. M. Billah, D. E. Porter, and I. V. Ramakrishnan. Sinter: Low-bandwidth remote access for the visually-impaired. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2016.

[10] J. Nichols, Z. Hua, and J. Barton. Highlight: a system for creating and deploying mobile web applications. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 249–258. ACM, 2008.

[11] J. Xu, Q. Cao, A. Prakash, A. Balasubramanian, and D. E. Porter. Uiwear: Easily adapting user interfaces for wearable devices. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, 2018.

[12] J. Yang and D. Wigdor. Panelrama: Enabling easy specification of cross-device web applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2783–2792, New York, NY, USA, 2014. ACM.