

A Toolkit for Usability Testing of Mobile Applications

Xiaoxiao Ma, Bo Yan, Guanling Chen, Chunhui Zhang, Ke Huang, and Jill Drury

Computer Science Department, University of Massachusetts Lowell
1 University Avenue, Lowell, Massachusetts, 01854
xma, byan, glchen, czhang, khuang, jdrury@cs.uml.edu

Abstract. The usability of mobile applications is critical for their adoption particularly because of the relatively small screen and awkward (sometimes virtual) keyboard, despite the recent advances of smartphones. Traditional laboratory-based usability testing is often tedious, expensive, and does not reflect real use cases. In this paper, we propose a toolkit that embeds into mobile applications the ability to automatically collect user interface (UI) events as the user interacts with the applications. The events are fine-grained and useful for quantified usability analysis. We have implemented the toolkit on Android devices and we evaluated the toolkit with a real deployed Android application by comparing event analysis (state-machine based) with traditional laboratory testing (expert based). The results show that our toolkit is effective at capturing detailed UI events for accurate usability analysis.

1 Introduction

Led by the rapid growth of the smartphone market, mobile Internet usage in the US is expected to approach 100% penetration and to reach 50% total usage by 2013 [1]. The usability of the mobile applications, however, remains a thorny issue. A recent study shows that the task completion rate using mobile Web ranges from 38% to 75% on different phones [2]. The average success rate was only 59%, substantially lower than the roughly 80% success rate when testing websites on a regular PC today. Another study shows that 73% of users experienced the slow-to-load problem when using the mobile Web, and 48% of users found mobile Web applications difficult to read and use [3].

In this paper, we focus on the usability testing of mobile applications, particularly native (instead of Web based) applications. We envision a system that can automatically collect user interface (UI) events as the user interacts with the mobile application. The collected UI data will then be uploaded to a remote server for either automated or manual usability analysis. This kind of system can complement traditional laboratory testing, and we believe it will be particularly useful to deploy for field-based usability testing. For many mobile application developers, it is often too costly to conduct extensive laboratory-based usability testing and we anticipate that the system described in this paper will be an

indispensable toolkit for low-cost usability analysis. We have implemented an Android-based automatic usability toolkit. To use our usability testing system, the Android developer needs to modify the application source code by inserting statements calling our library, which captures UI events and uploads them to a central server. We have designed the system to minimize the amount of required code modification and the impact of event-uploading overhead. To evaluate this system, we conducted a traditional laboratory-based usability testing on a home-built Android application, and compared it with state-machine based usability analysis using collected UI events. The results show that our usability toolkit can effectively capture most of the usability problems, some of which were not even discovered by traditional laboratory testing.

In the rest of this paper, we first discuss related work in Section II. Section III describes the details of the design and implementation of our toolkit. Then we discuss the user study and present the usability analysis results in Sections IV and V, respectively. Last we talk about some potential issues in Section VI and conclude in Section VII.

2 Related Work

Many studies have been done with event logging methods, which are compared to traditional laboratory testing methods in terms of usability problems identified. Tullis et al. [4] presented results that showed high correlations between laboratory and remote tests for task completion data and time-on-task data. The most critical usability issues with web sites were identified by both techniques, although each technique also uniquely uncovered other issues [5]. Another study by West and Lehman [6] was conducted to evaluate a method for usability testing with an automated data collection system. They found it to be an effective alternative to a laboratory-based test [5], but these studies were conducted on desktop machines instead of mobile devices.

Waterson et al. [7] conducted a remote usability study on mobile devices. They asked participants to find some information on a web site with wireless Internet-enabled digital assistants. Half of the participants ran the test in a traditional laboratory set-up while the other half performed the task with an observer present, but with an event logging tool to collect clickstream data remotely. They revealed that the event logging and analysis tool can easily gather many of the content-related usability issues, but had difficulty in capturing device-related usability problems. However, their study focused on the mobile websites rather than the mobile applications.

There have been few usability tools developed especially for mobile applications. Flurry Analytics¹ was developed to provide accurate, real time data to developers about how consumers use their mobile applications, as well as how applications are performing across different handsets. Application developers receive anonymous, aggregated usage and performance data, as well as the use

¹ <http://www.flurry.com/>

of robust reporting and analysis tools. However, this tool focuses on statistical information instead of identifying usability problems.

3 Design and Implementation

In this section, we first provide an overview of the Android UI framework, which forms the foundation for our event logging system. Then we discuss the details of its implementation and how it can be integrated with Android applications.

3.1 Android UI Framework

To set up the event logging system and integrate it with developers' applications, we need to have a comprehensive understanding on Android system's UI components, as well as how these components communicate with users' interaction. So here we give a brief introduction of this knowledge. The user interface of an Android application consists of *Activity* classes (terms with italic font indicates they are classes of Android Library; we use this convention throughout this paper, unless specially stated). Each screen in an Android application is a Java class that extends the Activity class, and activities use *Views* to form graphical user interfaces that display information and respond to user actions. An activity itself can be considered to be the root View for a screen and it is composed of smaller Views, which are interaction components like controls or widgets. Also, there might be other components attached to an activity, such as *Dialogs* or *Menus*, which are usually small windows that may appear in front of an activity. The top-most window in the foreground (View, Dialog or Menu) will always intercept and handle user inputs, whether it occupies a full or only partial screen.

To allow an application to react to user events such as a key press, screen touch and button click, developers have to utilize either event handlers or event listeners to accept user inputs and respond to them. An event handler is usually implemented by overriding some callback function while an event listener is a programming interface that contains a single callback method. As both event handlers and event listeners are constructed through calling underlying Android callback functions and they have the same functionality, for convenience we will use the term "Event Listener" to stand for both of them in the rest of this paper.

Usually an Event Listener is only attached to the window that registers it, and it will "consume" the events captured by it. This means if a button registers an *onClickListener* and then is clicked, the Event Listener of the button would intercept and handle the click, while the screen activity that owns the button has no idea that there was ever a click. Hence, if we want to log every single movement of users' interaction, for each Event Listener that handles a user input, we need to trace into the innermost level of the window that possesses it, and acquire our desired information that is intercepted by it.

3.2 Event Logging System

For usability studies of websites, it is possible to build an instrumented Web browser which allows the users' interactions with the websites to be automatically logged in a fashion that is transparent to the website developer. On the other hand, this is not feasible for Android applications because the UI interactions cannot be automatically captured. Application developers must get involved in modifying the source code and capturing UI events explicitly. To minimize developers' effort, we provide an event logging toolkit that takes care of most of the work of event capturing, formatting and packing, and transmission.

Our event logging system works as follows. The developers make small modifications to the source code of their applications adding API calls and recompiling the source code with the Software Development Kit (SDK) we provide. The SDK contains the APIs for each Event Listener and the developers call the corresponding API in their own listener code. The functionality of the Application Programming Interface (API) is to log ongoing user interaction events, its timestamp and properties of the relevant windows. For example, by inserting one statement in a View's *onClickListener*, the library can retrieve information such as the View's identifier, size, owner activity, and so on.

The recompiled applications now can automatically record the users' UI events, and transmit the captured interaction data periodically to a central server. These events are then used for usability analysis by the evaluators. Instead of transmitting the events immediately to the remote server, the logger runs in the background as a service and puts the captured events in a memory queue. When the number of events accumulates to a predetermined amount, they are transferred to our remote server through the 3G or WiFi network. If there is no available network at the time of transmission, these events will be stored on the device's Secure Digital(SD) card or hard disk sequentially according to their availability. In every event uploading cycle, these two places will be examined, and all existing data stored there will be transmitted if the network allows. The event transmitter creates a new thread for the transporting module so that this process is separated from the UI process.

The remote receiver module simply provides a relational database, as all the UI events are saved into different tables depending on their type. Usability analysis can then be conducted either manually or automatically (see Section V).

3.3 The Logger Implementation

Android has many Event Listeners, and one listener can be attached to a View, a Dialog or a Menu. As Views, Dialogs and Menus have different appearances and functionalities, they have different sets of listeners. For instance, clicking on a Menu may trigger *onMenuItemClickListener* while clicking on a View may trigger *onViewClickListener*, though both of the interactions are click events. Moreover, even if two subclasses inherit from the same parent, they may not have the same sets of listeners. Take View for example, the subclass *AbsListView* nests

the interface component *onScrollListener*, which will handle the user's scrolling on this view. In comparison, most subclasses of View do not support scroll events (they discard these events) if they do not implement *onScrollListener*. Indeed, the View class hierarchy is quite complex, as it has 8 direct subclasses and dozens of indirect subclasses. To assure we log user interaction events as completely as possible, we performed a thorough survey of the class hierarchy of View, Dialog and Menu, and only extracted those listeners that are related to user interaction. We then consolidated the listeners in the following ways.

View events Some listeners differ in their names or adhering classes, but they take care of the same user interaction, e.g. *onOptionItemSelectedListener*, *onMenuItemClickListener* and *onMenuItemSelectedListener* may all be responsible for selecting one item in an Option Menu. More interestingly, the three listeners can be registered at the same time, which means that they can all be triggered upon one click. Since we are interested in the user's interaction type rather than listeners' name, we decided to combine these sorts of similar listeners into one event type to avoid redundancy. Some listeners were distinguished by the type of user event they can handle but take identical parameters, such as the View's *onClickListener* and *onLongClickListener*, which deal with View's short click and long click events, respectively. For the simplicity of the logging library and back-end database design, we treated them as the same events as well, but differentiated them by adding a flag parameter. Thus the View's events were consolidated into *view click*, *view key press*, *view scroll* and *AdapterView item click* (note the terms for event types are also italicized). An *AdapterView* is a View whose children are determined by an *Adapter* object acting as a bridge between the *AdapterView* and its underlying data, e.g. an *ArrayAdapter* allows an array of objects to reside in the *AdapterView*.

Dialog and Menu Events triggered by Dialog and Menu are harder to capture. Dialogs and Menus are usually managed by their adhering activities, and they may not have an identifier (anonymous). Thus in order to locate which Dialog or Menu was fired, we need to infer it from the users' action sequence, by looking at which activity, through which action a Dialog or Menu, was initiated and how a Dialog or Menu was dismissed. It is not difficult to record the initiation event, but some attention needs to be paid to the dismiss event. Dialog has an *onDismissListener* but we cannot rely on it because it does not provide any information about how a Dialog was dismissed. Even worse, no matter how a dialog disappeared, this listener would always be triggered. For instance, if a user presses the OK button in a dialog, Android will first call Dialog's *onClickListener*, and then call *onDismissListener* as the dialog will disappear after the action. This will cause a double counting problem because the single event fires two listeners. Fortunately, we found that if a dialog was not dismissed by hitting the BACK key on the hard keyboard, at least one of the Dialog's *onClickListener* methods will be called, and it tells us which button was clicked that caused the dialog to be dismissed.

We have a different solution for Menu’s dismiss event. An *onOptionsMenuClosed* or *onContextMenuClosed* method will be toggled when a menu is closed, depending on the menu type. We can monitor *onMenuItemSelectedListener* to judge whether a menu was closed by selecting an item or by other means. Overall, we included *dialog key press* and *dialog button clicked* events for Dialog and *menu item select* and *menu close* events for Menu to our logging library.

System keys Android devices have BACK, MENU, HOME and SEARCH hard keys, and we name them system keys collectively as these keys function for all applications in general. Since an Android application is composed of activities, we can override *onKeyDown* listener in each activity to intercept these system keys. However, Android disabled HOME key interception to preserve its system’s running state. If the developers can intercept all key events, they can easily make the system malfunction, such as preventing users from quitting their applications by intercepting the HOME key press. Thus, we have to find other ways to detect that a user clicked the HOME key. Through a class named *ActivityManager*, we can acquire the currently running tasks as well as the top activity of each task. Then we can override each activity’s *onStop* method to check if its top activity equals to *com.android.launcher*, which is the activity name of the Android home screen, to decide if the activity is stopped by clicking the HOME button.

Unhandled events We believe that the above effort can already help us to record those events whose listeners were registered by the developers. This, however, is not enough. We want to collect a comprehensive set of the user’s interaction behaviors, but the developers will most likely only register listeners in which they are interested. Suppose a developer does not register an *onKeyDownListener* method in one activity, and a user tried to click a button that belongs to this activity. If, for some reason, this user missed clicking on the button but happened to click on the activity itself (recall an activity is also a view), the application will still run well because that activity will discard this mis-click by default. The developers may not care about this kind of event, but these events can be important to discover usability issues. For example, if we detected many clicks around the button, we can infer that the button is hard to click for some reason. Thus we would like to capture such events as much as possible, and we name them unhandled events in general.

In summary, the events we captured were classified by their adhering class and are listed in the Table 1. For different events, we obtained different attributes according to their own available properties. From an event that occurred in a View, we can retrieve its ID, width, height and its parent class etc., while for events that happened in a menu or dialog, they may not have such information. But for whatever window, we tried to retrieve as many event attributes as possible.

3.4 Code Revision

The extent of the revision that needs to be done in the developers’ code greatly depends on the hierarchical organization of the source code. Thus here we only

Table 1. Event Type Summary

	Event Type
View	click, key press, adapter item click, scroll
Dialog	key press, button click
Menu	item select, close
Other	unhandled motion event, unhandled key event, home key click, system key click, preference click

discuss the best and the worst cases in terms of the code modification workload. In the best case, all activities in an application extend from a single root activity, so the developer just needs to insert one event recording statement (by calling our API) into the *onTouchEvent*, *onKeyUp*, *onStop* methods of that root activity. Meanwhile, the event listeners were implemented uniformly rather than redefined in each View/Dialog/Menu, so that only one recording statement in the implemented View/Dialog/Menu event listeners needs to be inserted.

In the worst case, the activities in the application have no hierarchical structure at all and the event listeners were implemented separately from each View/Dialog/Menu. Then the developers have to insert the recording statement into the *onTouchEvent*, *onKeyUp*, *onStop* methods in each of their activities, and insert the recording statement into each View/Dialog/Menu Event Listener.

In both cases, we require that the application classes extend the application class of our own library so that our library can make use of the static memory space allocated to the application.

There are two additional challenges. First, the Android framework has already implemented some event listeners by default, such as the *onKeyDownListener* for an edit box. In this way, an edit box can accept key presses even without developers registering this listener explicitly. For such cases, we have to override the related listeners and register them so as not to miss recording the user's input. Second, the hierarchical relationship between classes can be troublesome for counting the user's interaction events accurately. While usually the user's interaction in one window will not be passed to its parent, the developers can call the super method to allow this to happen. If we add the logging function in both super and subclasses, one event that happened in the subclass may be recorded twice. Thus to avoid this double counting problem, we have to examine the application's class hierarchy and check whether at some point the developers called the super method.

In the future, we plan to provide a tool that can automatically inspect an application's source code and make appropriate changes for event logging, without the developers' involvement.

3.5 Events Not Captured

So far, our logging system is able to capture all the events that the developers have already set to listen as well as those unhandled events at the activity

level. But there still remain some events that are not being captured. First, we cannot log events that occurred in Android native activities, such as the Android Settings. We do not, however, really need these events as the goal of our system is to identify usability issues of the third-party applications, rather than the Android native screens. Second, we have not found any feasible way to trace keystrokes on the Android soft keyboard. Finally, we did not capture the unhandled events occurring in child Views, Dialogs and Menus of an activity. Although this is doable by registering listeners in each View/Dialog/Menu, we do not think it deserves so much effort (which involves changing source code) compared to its usefulness and potential event logging overhead.

Despite missing these events, the current system can already capture a comprehensive set of interaction events that can be used for usability analysis, as demonstrated in the next section.

4 User Studies

We conducted a user study to evaluate whether the proposed event logging toolkit is effective and helpful in identifying usability problems. One Android application called AppJoy [8], which has been deployed to the Google Market, was used as our subject application. It was developed by our group; we have the source code so it is convenient for us to integrate it with our logging library. We recruited participants to use this application and asked them to execute certain tasks assigned by us. Afterwards, we examined the logged events for usability analysis.

4.1 AppJoy Overview

The explosive growth of the mobile application market has made it a significant challenge for users to find interesting applications in crowded App Stores. While the application stores allow the users to search for applications by keywords or browse top applications in different categories, it is still difficult for the users to find interesting applications that they like. In light of this problem, existing industry solutions often use users' application download history or their ratings on some applications to predict users' preferences, such as Amazon's book recommendations. However, "downloading" is actually a weak indicator of users' fondness for an application, particularly if the application is free and users just want to try it out. Using application ratings, on the other hand, suffers from tedious manual input and potential data sparsity problems.

AppJoy makes personalized application recommendations by analyzing users' statistical usage of their installed applications. It can also allow users to browse popular applications according to their location, and track their application usage.

4.2 Participant Briefing

Participants were recruited through posters. We recruited 12 participants in total, all of them were undergraduate or graduate students of our school. We asked the participants to fill in a demographic information survey before the study. The questions included gender, major, own cellphone platform, familiarity with Android platform and previous experience in usability testing, and so on.

Among the participants, 7 were from the Computer Science department while 5 were not, 7 were male and the other 5 were female. All of them were between 20 and 35 years old. 2 participants owned an Android phone, 3 participants owned an iPhone and 7 participants did not have a smartphone. One iPhone user and one non-smartphone user also used an Android phone before, so in addition to the two participants who owned an Android phone, we had 4 Android phone users, but none of them were Android developers. 4 participants had previous experience with usability tests. None of the participants had used AppJoy before.

4.3 AppJoy Tasks

All participants were given the same Android device – Motorola Droid with Android version 2.2.2. AppJoy was preloaded onto the device and the device was connected to a university WiFi network. Every participant was assigned the following tasks one by one in the same order:

1. Browse recommended applications in AppJoy and “dislike” the first application the user is not interested in.
2. Browse the newest applications in AppJoy and install the first one whose rating is greater than 4 stars and the number of downloads is greater than 50,000.
3. Clear search history in AppJoy settings.
4. Search applications used by people in Boston, and point out the first application that was installed on this device.
5. Use AppJoy to uninstall the application that has been installed in task 2.
6. In AppJoy, look up the usage time of AppJoy.

As we did not code a special function to indicate the completion of a task, we used Android’s SCREEN_ON and SCREEN_OFF broadcasting events as the separator between tasks during the test. Participants were asked to give the device back to one of the evaluators after completing each task, and the evaluator turned off and turned on the screen twice and reset the application to its homepage before the next task.

5 Evaluation Results

In this section we answer the question of whether the UI events collected via the Android UI framework can indeed be used for usability analysis. The laboratory-based usability testing method known as formal usability testing is one of the most widely used approaches by usability practitioners for performing usability

Table 2. Usability Problems Identified by Laboratory Testing

	AppJoy Problem	Android Convention
Cosmetic	5	2
Minor	4	2
Major	4	2
Catastrophe	1	0

analysis [9, 10]. Thus we performed a laboratory-based usability test and compared it to a quantified state-machine based analysis using the collected events.

When the participants were executing tasks, we asked them to “think aloud” [11] and had several evaluators sitting beside them to take notes. At the same time, all of the user’s interaction events with the AppJoy were simultaneously logged and the data was transmitted to our server. In this way, we were able to get a fair comparison for the two different methods, since they were compared using the same participants, the same Android device, at the same time, within the same testing environment and the same user behaviors.

During the test, we lost two participants’ data due to the misconfiguration of the event logging system. So when comparing the two methods, we only considered information collected from the remaining 10 participants. The two people we lost data from were participant 4 and 5, one of them is male and the other is female. Neither of them majored in computer science and neither of them had prior Android experience. When presenting the evaluation in this section, we do use all 12 participants’ data except for the comparison results.

5.1 Traditional Laboratory-Based Usability Testing Results

When the participants were executing tasks, we asked them to “think aloud” and had 3 evaluators taking notes beside them. The evaluators were all very familiar with AppJoy and one of them is the lead developer. In order to get a better understanding of wrong moves the participants made when executing specific tasks, we talked with them about the difficulties they encountered during the test, and what caused their confusion. We found that these conversations with participants were indeed valuable for us to judge the exact cause of a usability problem. After the experiment was over, the evaluators discussed and consolidated usability problems identified based on their notes, the participants’ survey and their verbal feedback. Then we rated the severity of each usability problem according to Nielsen’s severity rating criteria [12], and summarized them in Table 2.

Some of the problems were apparently caused by the AppJoy design, which we call AppJoy problems. Some of the other issues could not be categorized as AppJoy problems because AppJoy just leveraged some components of the Android framework that caused the user confusion. For instance, some participants did not know how to view the notifications on Android, as they tapped on the notification bar instead of dragging. Also, there was one participant who said that he/she did not know how to scroll the view on the screen, and he/she moved

his/her finger in the opposite direction. For these problems, we say that they were generated by the users’ unfamiliarity with some of the Android conventions.

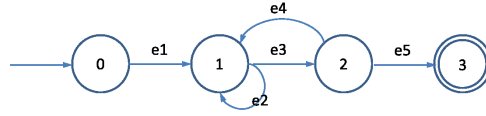
In addition, we have two problems not included in Table 2. One problem was that the participants had trouble finding the AppJoy setting. The reason for this problem was unclear, and we could not arbitrarily say whether this was because the participants did not know that pressing the Menu button can trigger application settings as an Android convention or they did not believe that the Menu is the right place to find the AppJoy settings. Although the confusion was mostly from the Non-Android participants, one Android participant also spent a lot of effort before getting to the right place. The other problem was that the participants frequently touched the AppJoy’s caption bar by tapping, dragging or scrolling. This problem is not negligible as 5 participants had this issue in 6 tasks. However, we cannot simply blame either AppJoy or Android for this as we do not see that any application or the Android framework itself defined the functionality of the caption bar. Thus we left the two problems described above uncategorized.

5.2 Event Logging Method Result

Hilbert and Redmiles described the sequence comparison technique as the process of comparing a target sequence against source sequences and producing measures of correspondence [13]. Here we leveraged the Finite State Machine (FSM) approach as our sequence comparison method for data collected from the event logging toolkit. We believe that FSM can maximumly make use of these data because the experiment was task-based: we can identify the *baseline* (correct) sequence for each task; and we have the participants’ entire clickstreams so we can examine how users’ interaction sequences deviated from the baseline sequence.

First we draw finite state machines with a baseline sequence for each task. These state machines only involve indispensable steps from the AppJoy home page to the destination state of task completion. Each step here is a state in the finite state machines, and the user actions are represented as transition functions between states. Figure 1 shows the baseline finite state machine for task 1, and Table 3 lists the associated activities corresponding to the states, and the user actions corresponding to the transition functions. As we can see, there are four states in this state machine with state 0 being the initial state and state 3 being the final state. Because this state machine represents the baseline sequence, we name the states in the state machine as fundamental states. Events e1, e3, e5 are the three imperative actions to complete this task, while e2 and e4, though not required, are also considered to be “correct” actions. Recall the task was asking the participants to dislike one application they were not interested in, so the participants may freely browse the applications by scrolling in the activity, or by checking the detailed information of an application and then going back. Events e2 and e4 correspond to these two actions, respectively.

Clearly not all participants will follow this baseline sequence, and they may enter many activities or states that are not shown in Figure 1. We call those

**Fig. 1.** Task 1 Baseline State Machine**Table 3.** States and Transitions for Baseline State Machine of Task 1

Activity		Event	
a0	Home Page	e1	Click <i>My Recommendations</i>
a1	My Recommendation	e2	Scroll in <i>My Recommendations</i>
a2	Application Detail	e3	Click in an Application
a3	Dislike Dialog	e4	Click <i>Back</i> Button
		e5	Click <i>Dislike</i> Button

states mistake states. We analyzed these mistake states and counted two kinds of behaviors that can, to some extent, indicate participants' confusion. One is backtracking (sometimes called "regressive behavior" by usability evaluators), and the other is engaging in unhandled events as we mentioned before.

Here we define backtracking as redundantly returning to the state that has already been traversed. We use "redundant" only to exclude situations where backward transitions are also considered to be appropriate responses, such as e4 in task 1. Basically if a user goes back to a state which he/she has visited before, that is backtracking, no matter whether he/she traces back from a baseline state or a mistake state. Additionally, if a user goes back from one activity to another, then immediately back to an even earlier state, we count this circumstance as two backtracking events rather than one. Usually backtracking reflects a user's confusion. When a backtracking event happens [14], it means that the user has picked the option that he/she thought to be most probably right, but apparently he/she did not reach the desired state via that option. For example, to find the usage time of AppJoy, many participants went to AppJoy settings first. After realizing there is no such information, they stepped back to the home page of AppJoy.

Similarly, unhandled events are user behaviors that occurred beyond the developers' expectations, since the developers did not even register listeners for those events. Although some events were triggered by the users' unintentional touches, most of these events reflected the users' intentional purpose. If a user performed a lot of such actions, we can infer that this user might not know where to navigate to the next step, as he/she was trying actions either randomly or exhaustively, hoping to hit something correct by chance or by systematically attempting to activate all plausible interface actions in turn. We list the number of instances that occurred for the above two behaviors by task in Table 4.

Simply from these numbers, we can infer that task 3 and task 6 were the two most difficult jobs for the participants, and tasks 1 and 2 were relatively easy.

Table 4. Backtracking and Unhandled Events in Each Task

	T1	T2	T3	T4	T5	T6
Backtracking	14	13	36	35	14	70
Unhandled Motion	1	4	11	1	1	21
Unhandled Key	7	3	4	1	30	4

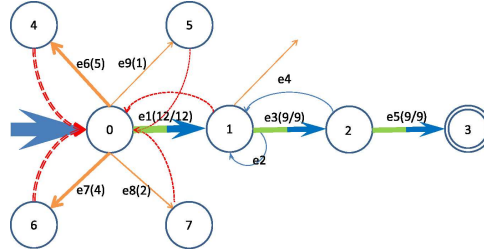


Fig. 2. Task 1 Traffic Flow Chart

Table 5. States and Transitions for All User Actions of Task 1

Activity	Event
a4 Most Recent	e6 Click <i>Most Recent</i>
a5 My Downloads	e7 Click <i>My Downloads</i>
a6 Location-based Search	e8 Click <i>Location-based Search</i>
a7 Top-Rated	e9 Click <i>Top-Rated</i>

However, having only these numbers is insufficient for us to analyze usability problems; thus we examined these events more closely.

As we recorded the participants’ every single move from one activity to another, we represent these transitions as well as the volume of these transitions graphically, in a form of traffic flow chart. We present task 1’s traffic flow chart in Figure 2 and list its states and transition functions in Table 5. Note here we only show those states and transition functions that were not included in Table 3 (in other words, mistake states and transition functions).

We use the width of an edge to represent the volume of the transitions. Blue edges represent incoming flows, while green edges and orange edges represent outgoing flows to a baseline state or a mistake state accordingly. Additionally, we use dashed red edges to indicate backtracking flows. The volume of traverses along with each transition function is marked above each edge. Note that the number of correct outgoing flows from one baseline state and the number of incoming flows to the next baseline state have different implications. The former means how many times the participants made the right choices from one state, and the latter means how many situations occurred in which the participants were asked to make a choice. For example, if one participant went to some mistake state from state 1, and then returned back to state 1, the number of incoming flows to state 1 would be greater than the number of correct outgoing flows from state 0. Thus we have two figures above every baseline transition

function, with the first one representing correct outgoing flows from one state and the second one representing incoming flows to the next state. Coincidentally, the two figures for all baseline transitions turn out to be equal in this task.

As shown in Figure 6, the participants traversed to the home page of AppJoy 24 times, of which they went to state 1 12 times, while they went to different mistake states another 12 times. Comparatively speaking, from state 1 to state 2, among the 12 incoming flows, 9 times the participants progressed immediately to state 2, backtracking occurred two times during this stage, and one participant failed to find the next move. From state 2 to state 3, all of the 9 incoming flows traveled to the correct state. The correct flow ratios for the three stages were 50%, 75% and 100%, respectively. Clearly the users were less confused at state 2 than at state 0. We calculated the success ratio of flows in each step as a measurement for detecting usability problems, because this can be considered to be a measure of the users' confusion at each step. Also, the amount of flow that entered the final state is actually the number of participants who successfully completed a task. On the other hand, the number of participants who failed to complete a step is also an important indicator of usability problems. We calculated these metrics for each task and summarized them in Table 6.

Note that the step in which backtracking occurred is determined by the next baseline state rather than the current baseline state, as we think users' difficulties in locating the next baseline state is the main reason that causes backtracking in the current baseline state. For instance, the 12 backtracking events that happened during the transition between state 0 and state 1 were counted as backtracking events that occurred at state 1.

Many usability problems can be discovered by reviewing Table 6. Too much backtracking, such as the amount that occurred at step 1 of task 6, indicates that the desired information is located at a different place than anticipated by participants, or it was not visible to users. Low correct flow ratio, in our case less than 50%, is another sign of potential usability problems. For instance, the ratio of correct flows for step 1 of task 3 was only 28.6%, which is congruent with our previous discussion regarding participants having problems finding AppJoy settings. Also, if the number of participants who cannot complete a step exceeded a certain threshold, there is possibly a usability problem. For instance, 4 out of 10 participants failed to find the usage time at step 2 of task 6. Although we are unaware of the cause, we would strongly recommend that the developer of AppJoy inspect that component to show usage time clearer more clearly.

By taking a step further to examine which mistake state attracted most of the incorrect traffic, we may possibly predict the reason for that problem. If many participants went often to the same mistake state, that state must be very confusing to the participants. Hence we call that state the "most misleading state". For example, at step 3 of task 4, 18 out of the 20 incorrect flows went to the detailed page of location-based applications; obviously this is the most misleading state. We guess the underlying reason of this problem is that the users sought to find something at the summary screen of the location-based applications, but that information was not sufficiently visible.

Table 6. Traffic Flow Metrics Based on Number of Mistakes

Metrics	Step 1					
	T1	T2	T3	T4	T5	T6
No. of Backtracks (*)	12	10	29	11	11	61
Mistake State No.	4	3	12	6	4	8
Correct/Incorrect Flows	12/12	12/10	16/40	12/7	9/12	14/58
Correct Flow Ratio	50%	54.5%	28.6%	63.2%	42.9%	24.1%
Fail to Pass No.	1	0	1	1	3	0
Metrics	Step 2					
	T1	T2	T3	T4	T5	T6
No. of Backtracks	2	3	7	5	2	9
Mistake State No.	0	0	1	2	1	3
Correct/Incorrect Flows	9/3	13/3	9/7	11/6	11/2	6/8
Correct Flow Ratio	75%	81.3%	56.3%	64.7%	84.6%	42.9%
Fail to Pass No.	0	0	0	1	0	4
Metrics	Step 3					
	T1	T2	T3	T4	T5	T6
No. of Backtracks	0	0	0	19	1	N/A
Mistake State No.	0	0	0	2	0	N/A
Correct/Incorrect Flows	9/0	10/0	9/0	8/20	10/1	N/A
Correct Flow Ratio	100%	100%	100%	28.6%	90.9%	N/A
Fail to Pass No.	0	0	0	0	0	N/A
Metrics	Step 4					
	T1	T2	T3	T4	T5	T6
No. of Backtracks	N/A	N/A	0	N/A	N/A	N/A
Mistake State No.	N/A	N/A	0	N/A	N/A	N/A
Correct/Incorrect Flows	N/A	N/A	9/0	N/A	N/A	N/A
Correct Flow Ratio	N/A	N/A	100%	N/A	N/A	N/A
Fail to Pass No.	N/A	N/A	0	N/A	N/A	N/A

* No. of Backtracks = number of backtracking incidents

Table 7. Traffic Flow Metrics Based on Number of Participants Making Mistakes

Metrics	Step 1						Step 2					
	T1	T2	T3	T4	T5	T6	T1	T2	T3	T4	T5	T6
No. of Backtracks	4	6	4	2	6	8	0	0	2	1	2	3
No. to the MMS (*)	3	2	5	2	7	8	0	0	0	2	1	2
Metrics	Step 3						Step 4					
	T1	T2	T3	T4	T5	T6	T1	T2	T3	T4	T5	T6
No. of Backtracks	0	0	0	4	1	N/A	N/A	N/A	0	N/A	N/A	N/A
No. to the MMS	0	0	0	4	0	N/A	N/A	N/A	0	N/A	N/A	N/A

* MMS = Most Misleading State

An interesting phenomenon emerged in task 5. Three participants failed to progress through step 1 but surprisingly all of the participants successfully completed this task. This means that some participants avoided one fundamental state and reached the objective state through another route. We provided this

information to the developer of AppJoy who confirmed that the way it was designed had some problems.

For unhandled motion events, we inspected the activities in which unhandled motion events occurred and their physical positions on the screen. Among 39 unhandled motion events across all tasks, 30 of them (76.9%) were clicks or moves at the caption bar in different activities; this was the case for all 11 events in task 3 and 17 out of 21 events in task 6. Although we didn't classify it as a usability problem, this phenomenon reflects the participants' frustration. As these behaviors happened frequently in the two tasks that participants had trouble dealing with. Maybe only when the users could not find other ways to complete a task, they touched the caption bar as a last resort. For the remaining 9 unhandled events, 8 of them were clicks or moves on blank pages between two activities; these actions were not noticed in laboratory-based usability testing. Even though we cannot conclude that lengthy loading time between two pages caused the users to perform such actions, at least this is an interesting finding that we did not expect: that such events can be captured by the event logging library. The last unhandled motion event was a mis-click on the side of one button that one participant intended to click, but this was a rare case.

For unhandled key events, we inspected the key code for each click and its related activity. Our library does not log the key code for keystrokes on the Android keyboard except for the delete key out of concern for the users' privacy. However, none of these unhandled key events were from keystrokes on the Android keyboard. Actually, only four keys were pressed: the volume down, volume up and camera focus keys on the right side of the device, and the search key below the screen (we did not classify the search key as a system key because most third party applications do not respond to this key). Except for the search key presses, other unhandled key events were not observed by the evaluators during the experiment. We speculate that these keys were probably pressed without the users' conscious purpose as these keys can be easily touched by mistake in daily usage. However, the search key presses constitute a different case, as 36 out of 49 unhandled key events are from search key presses, including all of the 30 unhandled key events in task 5. Recall that in task 5 we asked participants to uninstall the application they just installed in task 2; we can infer that in task 5, the users were trying to search the application directly with the search key because they knew the application name, but apparently AppJoy did not handle this key event. This confirms a usability problem identified in laboratory testing that AppJoy should add some mechanisms to search or at least sort the recommended applications. Furthermore, 3 search key presses were from task 3 and 3 were from task 6, yet these two tasks did not involve looking for a special application. We guess that the participants intended to search the functionalities of AppJoy setting and usage time by doing so.

The above statistics only dealt with the number of total occurrences of each event but ignored the differences between the participants. The number may be difficult to interpret if a minority of the participants made a large number of mistakes. In fact, if only a few people had problems with a user interface

Table 8. Usability Problems Identified by Auto Logging Method

	AppJoy Problem	Android Convention
Cosmetic	0(5)	1(2)
Minor	4(4)	1(2)
Major	5(4)	1(2)
Catastrophe	1(1)	0(0)

component, this component’s design is probably satisfactory. To avoid the difficulties of interpretation, we calculated how many participants backtracked and how many participants entered the most misleading state in each step of a task. In this way, we can alleviate the above issue: if the backtracking number for all participants was large but the number of participants who experienced backtracking was small, it means that a few participants were confused. Similarly, only if many participants backtracked from a particular state can we conclude that the state was really misleading. The two metrics related to the number of participants are summarized in the Table 7.

Regarding backtracking, the data in Table 6 and Table 7 shows fairly uniform behavior across participants, because more backtracking across all participants corresponds to more participants navigating with backtracking. Regarding the correct flow ratio and most misleading state, while the data again shows fairly uniform behavior, there are some minor differences that can be observed. For instance, at step 3 of task 4, the number of incorrect flows was higher across all users, compared to that at step 1 of task 5, but fewer participants went to the most misleading state. This means the problem in task 5 is more general across the users, hence that problem should be considered to be more critical. In summary, we can look at the data in both of the tables to rate the severity level of usability problems.

Besides the above measurements, developers or evaluators can almost “re-play” the users’ behaviors if they have time to manually review the logged events. Though time-consuming, this approach can help to detect additional usability problems, even for some subtle issues that were overlooked during the laboratory-based usability testing. For example, there was a button that overlapped with recommended applications which made it hard to be seen, so that only one participant clicked that button and none of the evaluators observed this phenomenon during the experiment. But by examining the logged sequence, we noticed this event and confirmed that this was a usability problem.

5.3 Comparison Results

Finally, we summed up all usability problems identified through the event logging toolkit, and compared the number with that discovered by laboratory-based usability testing in Table 8. Note that in this comparison we excluded the two participants’ data that was lost with the event logging method.

Usability problems identified by laboratory-based usability testing are shown in parentheses here for comparison. From the rest of the 10 participants, we iden-

tified exactly the same number of usability problems as from all 12 participants in laboratory-based usability testing. We can easily see that the laboratory-based testing method can identify more cosmetic problems. All 5 cosmetic usability issues observed through laboratory testing were not discovered by the event logging method. But the event logging method is effective for identifying critical usability issues, including major and catastrophic usability problems. All critical usability problems discovered through laboratory testing were found by the event logging method, and by manually reviewing the participants' behavior sequences, we found out one more major problem that was overlooked in the experiment. However, the shortcoming of the event logging method is, for most of the issues it identified, that although it can point out the location of a problem, it cannot tell the cause of that problem. This is an issue common to all event logging methods because they lack the information that can be gleaned from listening to participants' verbalized thoughts and observing participants' facial expressions and other nonverbal signals.

Compared to laboratory testing, the event logging method found fewer problems that were introduced by the users' unfamiliarity with Android conventions. We expected this because the library cannot record the users' interactions outside of AppJoy. But because the objective of this library is to find usability problems in third-party applications rather than in the Android framework itself, we do not consider this to be a big issue.

6 Discussion

We have only tested the event logging toolkit on one application, which is of course far from enough to conclude that it can be effective to help evaluators, developers and designers identify usability issues on all Android applications. We will integrate this library into more Android applications to validate its usefulness in the future. One thing to note is that our toolkit does not help applications developed without the Android UI framework, such as games based on OpenGL.

On the other hand, even after conducting just one test, we can already demonstrate that the proposed event logging toolkit can detect some subtle actions that are difficult to observe in laboratory testing, such as some quick moves and the unhandled events discussed above. Meanwhile, it can provide strong quantitative measurement and lots of statistical data describing users' interactions with the application. So it can at least complement traditional laboratory-based usability testing.

We tested the application in a WiFi network environment, which neglects possible networking problems that could happen under poor network conditions. Although we know that AppJoy sometimes has trouble connecting to the server under the 3G network, we did not identify this problem through this experiment. Hence we can see context information is needed to locate usability issues under some conditions, and this is precisely the weak point of laboratory-based usability testing. We will include context information retrieved from sensors of the

Android device in the next version of the event logging toolkit. Because we can collect the users' interaction data in a real world environment, we can determine usability problems under different conditions through the toolkit. We anticipate that this approach will be a big advantage for an event logging method as it is more suitable for field-based usability testing.

7 Conclusion

It is challenging to conduct usability testing for mobile applications. In this paper, we present a UI event logging toolkit that can be embedded into Android applications. The toolkit requires minimum source code modification by the developers and automatically uploads fine-grained UI events to a central server. By testing a deployed Android application, a state-machine based sequence analysis is evaluated using the logged events and compared to traditional laboratory-based usability testing. The results show that the proposed toolkit is effectively capturing detailed interaction events, which can provide accurate and quantitative analysis of usability problems. In summary, the event logging toolkit can discover most usability problems comparable to those uncovered by the laboratory method, and also reveal some unexpected issues. In future work, we will extend the toolkit so it can be deployed for field-based journal usability testing.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 1016823. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. R. Kerr, "Us mobile internet usage to near 50% in 2013," *Vator News*, August 2009.
2. R. Budiu and J. Nielsen, "Usability of mobile websites: 85 design guidelines for improving access to web-based content and services through mobile devices," *Nielsen Norman Group Research Report*, July 2009.
3. "Why the mobile web is disappointing end-users," *Equation Research Report*, October 2009.
4. T. Tullis, S. Fleischman, M. McNulty, C. Cianchette, and M. Bergel, "An empirical comparison of lab and remote usability testing of web sites," in *Usability Professional Association Conference*, Orlando, 2002.
5. J. M. C. Bastien, "Usability testing: a review of some methodological and technical aspects of the method," *Computing Research Repository*, vol. 79, pp. 18–23, 2010.

6. R. West and K. Lehman, "Automated summative usability studies: an empirical evaluation," in *Proceedings of the SIGCHI conference on Human Factors in computing systems*, ser. CHI '06. New York, NY, USA: ACM, 2006, pp. 631–639.
7. S. Waterson and J. A. L, "In the lab and out in the wild: remote web usability testing for mobile devices," in *Conference on Human Factors in Computing Systems*, 2002, pp. 296–297.
8. B. Yan and G. Chen, "Appjoy: personalized mobile application discovery," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, ser. MobiSys '11. New York, NY, USA: ACM, 2011, pp. 113–126. [Online]. Available: <http://doi.acm.org/10.1145/1999995.2000007>
9. S. Rosenbaum, J. Rohn, and J. Humburg, "A toolkit for strategic usability: results from workshops, panels, and surveys," in *Proceedings of the ACM CHI 2000 Conference on Human Factors in Computing Systems*, New York, 2000, pp. 337–344.
10. "Upa 2007 salary survey," *Usability Professionals' Association*, 2008.
11. K. A. Ericsson and H. A. Simon, "Verbal reports as data," *Psychological Review*, vol. 87, pp. 215–251, 1980.
12. J. Nieslen, "Severity ratings for usability problems," in *Retrieved June 4th from UseIt*, 2007. [Online]. Available: <http://www.useit.com/papers/heuristic/severityrating.html>
13. D. M. Hilbert and D. F. Redmiles, "Extracting usability information from user interface events," *ACM Comput. Surv.*, vol. 32, pp. 384–421, December 2000.
14. D. Akers, "Backtracking events as indicators of software usability problems," *Ph.D. dissertation*, 2009.