Doppio: Tracking UI Flows and Code Changes for App Development

Pei-Yu (Peggy) Chi, Sen-Po Hu, Yang Li
Google Inc.
{peggychi, senpo, liyang}@google.com

ABSTRACT
Developing interactive systems often involves a large set of callback functions for handling user interaction, which makes it challenging to manage UI behaviors, create descriptive documentation, and track code revisions. We developed Doppio, a tool that automatically tracks and visualizes UI flows and their changes based on source code. For each input event listener of a widget, e.g., `onClick` of an Android View class, Doppio captures and associates its UI output from a program execution with its code snippet from the codebase. It automatically generates a screenflow diagram organized by the callback methods and interaction flow, where developers can review the code and UI revisions interactively. Doppio, as an IDE plugin, is seamlessly integrated into a common development workflow. Our studies show that our tool is able to generate quality visual documentation and helped participants understand unfamiliar source code and track changes.

Author Keywords
Software documentation; IDEs; screenflow diagram; Android; mobile apps; screencast videos; demonstrations.

ACM Classification Keywords
H.5.2. User Interfaces — prototyping, input devices and strategies, graphical user interfaces.

INTRODUCTION
Interactive systems, such as a mobile or web application, often heavily involve dynamic visual behaviors in response to user input, including complex UI changes and animated feedback. To manage UI behaviors and track revisions, it is common that developers rely on software documentation, which describes the purpose and behavior of a system and its source code in human understandable language. Documentation serves a crucial role to help developers comprehend *what* the program is intended to achieve and *how* each element behaves [28] and further modify it.

CHI 2018, April 21–26, 2018, Montreal, QC, Canada
© 2018 Copyright is held by the owner/author(s).
ACM ISBN 978-1-4503-5620-6/18/04.
https://doi.org/10.1145/3173574.3174029

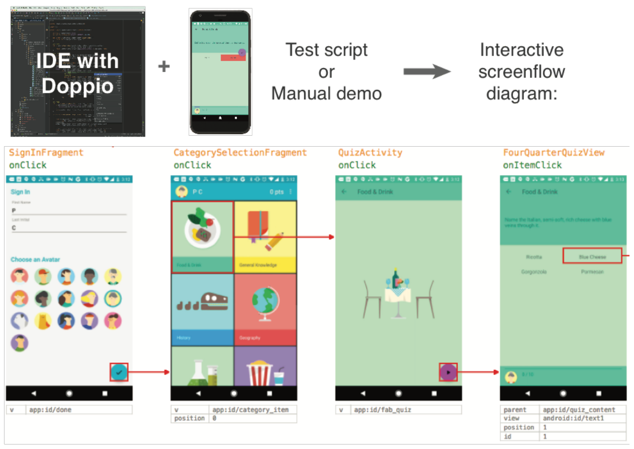


Figure 1. Doppio is an IDE plugin that enables developers to review their programs and track UI revisions with a callback-based screenflow diagram. It presents the screenshots, video clips, code snippets, UI regions, and run-time argument values for each user input, such as a click event.

However, writing documentation is time-consuming, and developers tend to take it on in haste, which results in incomplete, imprecise, or outdated content [42].

Previous work has investigated automatic approaches for generating text documentation from source code [39, 49, 58] or program execution [40]. Nevertheless, it can be difficult to describe visual, interactive behaviors in words. As a result, developers often resort to providing screenshots, animated GIFs, or screen-recording videos in documentation (e.g., README files or code change requests) to visually describe these behaviors. Again, this method requires manual creation of multimedia materials in a separate process from code development environment. Developers have to carefully select representative states of a program, which incurs enormous efforts and can be challenging to maintain as a program evolves.

We introduce Doppio, ***D**emonstration **o**f a**pp**l**i**cation **I**/**O***, an IDE tool that automatically creates callback-based documentation for visualizing the interaction UI flow and revisions (see Figure 1). Doppio does not require any additional effort from developers other than running and testing the program as they would typical do in a common development workflow (see Figure 2). As a first step, without loss of generality, our work is designed to enhance an Android IDE for mobile development. During a test session, Doppio monitors the behavior of each input event listener defined by the Android framework in the background, e.g., `onClick` of a `View` class. It automatically intercepts the invocation of a method and identifies the change on the UI from the runtime execution of the program. Based on the captured data, Doppio then analyzes the user source project, segments a screencast video, and generates useful information such as relevant screenshots, video clips, and code snippets shown in a screenflow diagram for interactive review. It also inserts a new Javadoc tag "`@look`" to the block comment of each captured method in source code that links to the corresponding screen states. These enhancements enable developers to easily inspect the behavior—*abstract* logic—of each individual method with *concrete* runtime examples.

To find out how well Doppio extracts visual demonstration, we tested it on 10 open source Android projects and investigated its usability with 16 professional Android developers. Our experiments indicate that Doppio effectively helped participants understand unfamiliar source code and further make modifications. In particular, our work makes the following contributions:

- An automatic approach for tracking methods' behaviors of an interactive program, by automatic code structure analysis, runtime method interception, and screencast video analysis from a test session.
- A novel visual documentation presenting an interaction flow of a mobile application based on its input sequence and callback functions.
- An integrated solution that enhances a mainstream IDE and supports developers' existing workflow of application development.
- A set of experiments that examined the feasibility of this approach, and insights into future directions.

RELATED WORK

We discuss three research topics that our work touches on, including tools for visualizing program execution, generating software documentation, and capturing user workflow from demonstrations.

Interactive Program Visualization

Researchers have suggested that by visualizing aspects of source code and enabling direct manipulation of code components, software tools can lower the complexity of programming [51, 43] and reduce time of code understanding [1]. There have been a variety of software visualization techniques integrated into code editors [15]. Examples include showing concrete examples and documentation [54], call graph structures [35], effects of user action [44] or camera input [36, 37], changes with code highlights [57], and interaction flows among multiple interactive devices [10]. Editors can also be enhanced to provide on-demand [8] or context-sensitive supports [46] and runtime value visualization [34]. Inspired by these approaches, Doppio assists developers in code understanding by presenting visual examples, but we focus on the visual traces of UI state changes and their code snippets based on input methods in order to connect abstract source code with runtime visual examples.

Prior research has investigated the benefits of interactive version control for code or design revisions [41]. Methods include managing alternatives [30] and through a design gallery [29], navigating in code history [59, 61, 32], and preserving editable code history [18] or micro changes [50]. Doppio shares the same vision through automatic code tracing for reviewing key changes, but we do not claim our contribution on a novel versioning tool overall.

Software Documentation Enhancement

Documentation is important for developers to comprehend and maintain source code [28], especially for large-scale software projects. Popular languages have defined formats for describing code behaviors. For Java, comments of a method, typically textual, include parts such as input (a list of arguments, each denoted by the tag "`@param`"), output (return values by the tag "`@return`"), and summaries that can be automatically transformed into documentation [55].

However, writing comments is a manual, time-consuming task. Developers are often reluctant to spend effort on documentation, which results in incomplete and outdated code comments [42]. Previous work has been devoted to tools for automatic documentation generation. Method behaviors can be summarized from source code [58], comments [39], and application context [49]. For type-loose languages such as JavaScript, types can be captured from program execution to complete documentation [40]. Code comments can embed information beyond text, such as multi-media material recorded by developers [27]. In the same vein, our work follows the paradigm of automatic approaches to enhance documentation. However, we focus on a unique aspect—generating multimedia demonstrations that can visually describe the code logic and behavior—by capturing the UI states of a program affected by specific methods in the program from its execution.

Workflow Capturing

A significant body of previous work provides techniques to record user demonstration and replay program states, often through program tracing or reverse engineering source code, such as for debugging [3], code understanding [4, 53], reusing web pages [6, 47], or testing mobile apps [56]. In particular, Whyline visualizes a captured trace for reasoning system behaviors after the program execution [38]. Doppio also adopts a post mortem approach, but while Whyline traces every detail for general-purpose reasoning, we focus on tracing a specific set of code elements for input-based visualization. Another project, Unravel, allows users to review and replay method calls and DOM changes of a website in a web browser [31]. It injects an observation agent into a site to track DOM differences during the recording phase. Doppio employs a similar approach by intercepting a program without functional interference.

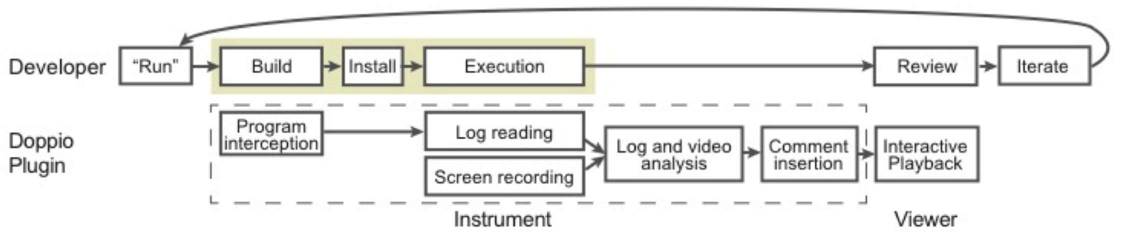


Figure 2. The Doppio pipeline: In an IDE, developers hit "run" to test the program. The IDE builds the app that weaves our interception logic and installs on the device or emulator. When the app is running, Doppio logs the execution details and records the device screen. Once the test is finished, it analyzes the captured information and renders interactive documentation.

However, Doppio is designed based on a fundamental understanding of the target system framework (Android). Instead of reverse engineering source code from recording, our tool analyzes code structures and provides *accurate* code tracing, which is critical for developers. We focus on capturing UI state changes and visible on-screen behaviors (from a screencast video) to present behaviors in a diagram. These aspects distinguish Doppio from the prior art.

Finally, Doppio is built upon previous work on capturing and presenting software UIs from user demonstration for workflow understanding. By recording user input events (e.g., a mouse click), useful information can be extracted from the screen pixels of an application, such as detailed views of manipulating an image or 3D model [24, 9, 7] or macro actions [60]. Visual summaries have been shown effective in learning unfamiliar concepts [48], and videos are helpful for understanding software behaviors [25, 26], especially for continuous operation or animation [9]. Doppio provides an integrated solution by capturing events, code, and screencast videos and allowing programmers to interactively review method-specific images and videos.

DESIGN GOALS

Based on our experiences with Android development and discussions with app developers, we identify three objectives that guide our design decisions.

Lightweight Workflow. For Android applications, developers commonly use an IDE (e.g., Android Studio [19]) to manage a project, write code, and deploy the app to devices or emulators for testing. A tool should support the existing workflow and minimize additional efforts from developers.

Complete Code Coverage. A highly-complex project may contain a large number of classes and methods, which imply complicated dependencies and relationships in code. A tool should cover the codebase completely based on a good understanding about the project and the underlying framework. Ideally, it should not rely on developers to specify code segments of interests.

Code-Centric Interactive Playback. To help developers focus on the code and its behavior, a tool should understand the code structure and provide information based on the software project. It should respond to the specific point of the document that developers are interested in.

USING DOPPIO

Doppio is a tool that automatically tracks and documents UI behaviors of a target interactive program based on the corresponding source code elements. We designed Doppio as an IDE plugin to enhance the current development workflow (see Figure 2). To discuss the detailed workflow with Doppio, assume a developer, Marilyn, is developing an Android application for providing quizzes[1] for her students using the Android Studio IDE with the Doppio plugin. Similar to common mobile applications, her application provides a main menu where users can select different question types. It also includes interactive components, such as buttons, sliders, checkboxes, and text fields for answering questions.

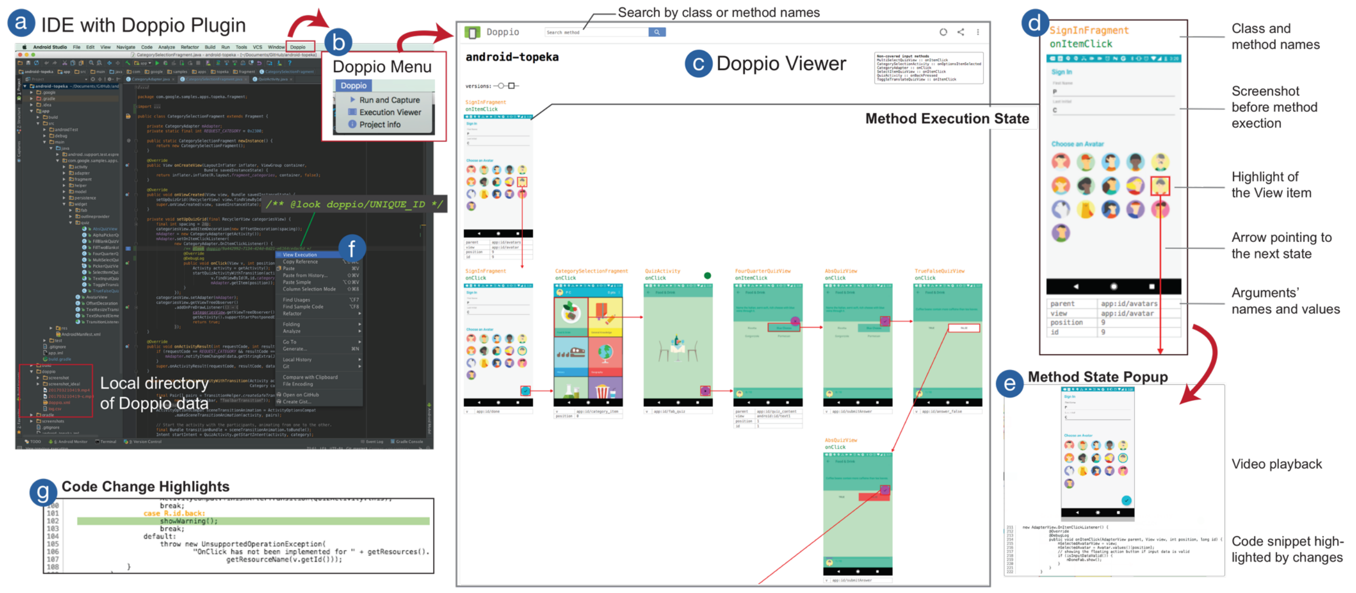
Capturing Application Execution

With her Java-based project opened in Android Studio (see Figure 3a), Marilyn opens the menu bar of the IDE and selects "*Run & Capture*" provided by the Doppio plugin (see Figure 3b). This is similar to what she usually does to "*run*" or "*debug*" her program on a connected mobile device. To test the application, Doppio flexibly supports either a session where developers manually provide the program input or a test session where the interaction flow is automatically executed by a pre-captured or pre-defined script, such as using the Android Espresso testing framework[2] [21]. Assume Marilyn runs a test script to avoid repetitive manual input in her development process.

After the application is installed and brought up on the mobile device, Marilyn sees a notification "*Doppio starts recording*" in the IDE. Our *Instrument* automatically records a screencast video of the target device to capture all the screen activities. Internally, it identifies and intercepts UI methods based on an understanding of Android code structure without the developer noticing the underlying mechanism. At compile time, the Doppio Gradle plugin traverses the Android program and looks for a specific set of UI methods that we identify, such as `onClick` of a `View`

[1] This example is inspired by the open source sample "android-topeka" by Google [22] that we also present in Figure 1 and 3.

[2] To support motion design and make interaction visible, we require a time gap between input actions in a test script via `Thread.sleep(long millis)`. This gap can be set depending on the app design. We recommend at least 3,000 milliseconds to ensure UI transitions are completely rendered.


a IDE with Doppio Plugin
b Doppio Menu
Run and Capture
Execution Viewer
Project info
/** @look doppio/UNIQUE_ID */
f
Local directory of Doppio data
Search by class or method names
Doppio
android-topeka
versions:
c Doppio Viewer
Method Execution State
d SignInFragment onItemClick
Class and method names
Screenshot before method exection
Highlight of the View item
Arrow pointing to the next state
Arguments' names and values
e Method State Popup
Video playback
Code snippet highlighted by changes
g Code Change Highlights

```
                break;
            case R.id.back:
                showWarning();
                break;
            default:
                throw new UnsupportedOperationException(
                        "OnClick has not been implemented for " + getResources().
                                getResourceName(v.getId()));
        }
```

parent	app:id/avatars
view	app:id/avatar
position	9
id	9

Figure 3. In an IDE (a), developers "run and capture" an app via the plugin menu (b). Execution will be logged, analyzed, and presented in the Doppio's Web Viewer (c) as an interactive screenflow diagram. The interaction flow and execution details are organized by callback methods (d), with the video and code snippet attached (e) and code change highlighted (g). Developers can review and search in this Viewer or from the source code in the Editor (f).

class. In this way, while the test input, such as clicking on a button, triggers an intercepted method call, information about the runtime behavior is recorded. If needed, Marilyn can manually annotate and monitor any method of her interest with a Java annotation "`@capture`", which would be useful for system-driven UI changes.

When the test session finishes, Doppio automatically analyzes the execution logs and segments the screen recording. Once the process is completed, a notification "*Doppio: playback ready (7.2 seconds used)*" shows in the IDE, and meanwhile the viewer is automatically launched in a Web browser. The video and its metadata are stored in developers' source code repository locally.

Reviewing Method Behaviors

In the Doppio's Viewer (see Figure 3c), Marilyn sees an overview of interaction flow captured from her test session. For each user input (such as clicking on a View item), Doppio shows the screen state *before* the method execution. On the screenshot, the target View is highlighted, with an arrow pointing from this item to the next screen state. Each state also shows the details that respond to the interaction, including the class and method names and its runtime argument values (see Figure 3d). When the mouse hovers over the state, the corresponding code snippet and its video clip is shown (see Figure 3e), where video can be replayed with a mouse click. These help Marilyn quickly verify the app behaviors visually at a glance.

When Marilyn navigates source code of her project back in the IDE, she notices that the methods invoked by the test session were tagged by Doppio in the comment blocks with a Javadoc tag "`@look`" (see Figure 3f). An icon next to the code line helps her visually identify these links. When she right-clicks on the link, corresponding screen states to the method are highlighted in the Viewer, from which she can also search by keywords. This assists her examining the method's abstract logic in the context of the source project.

Iterating App Design

After reviewing the execution, Marilyn decides to modify her code to enhance one specific user interaction. She adds a confirm dialog in the click callback of the "submit" button. She reruns the test script of the app using Doppio and then sees a new screenflow diagram in the Viewer, with a revision indicator "`versions:`–○–□–" that allows her to compare the revisions. The changed state and code different from the last version are highlighted, which helps her focus on the UI revision (see Figure 3g). For any changed code that Doppio tracks but is not captured in a test session, our tool provides a warning message to developers.

THE DOPPIO SYSTEM

We elaborate on the underlying mechanisms of Doppio pipeline (see Figure 2) and the system components (see Figure 4) that realize the presented scenario.

UI Event Logging by Program Interception

Android applications respond to user input events through event listeners. In the Android API, the `View` class defines event listener interfaces, such as `OnClickListener`. Developers, who follow the Model-View-Controller (MVC) design pattern, usually implement and register custom event

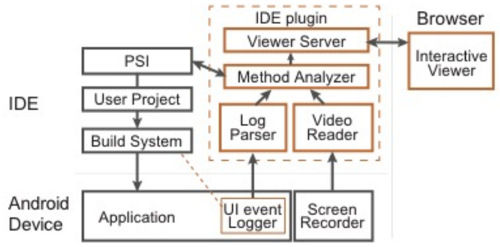


Figure 4. Doppio's system architecture, including an IDE plugin and a build system plugin in users' application. The colored blocks show the Doppio components.

listeners in controller classes, which typically extend Android's `Activity` or `Fragment` class. The following code snippet, extracted from a class in an open source project [22], shows how an extended `Activity` defines the logic:

```
1   View.OnClickListener mOnClickListener =
2     new View.OnClickListener() {
3       @Override
4       public void onClick(final View v) {
5         // Executes business logic for a click event.
6       }};
7   View mBackButton = findViewById(R.id.back);
8   mBackButton.setOnClickListener(mOnClickListener);
```

Here, Line 1-2 create a listener for click events. The application behavior responding to user input is defined inside the callback method in Line 4. Then, Line 7 finds a specific View object by the Android resource ID (the back button in this example), and Line 8 registers the listener to the button object. In the real world, mobile applications often consist of a large number of input callbacks. It requires a significant amount of time and effort of developers to trace source code and test run an application in order to gain a thorough understanding about the UI behavior associated with each event handler.

Based on this event handling framework, our goal is to capture UI event logging about timing and details of each method invocation, which serves three purposes: 1) segmenting a screencast video into snippets that show the corresponding UI events, 2) identifying the corresponding code, and 3) presenting useful execution information to developers interactively. Table 1 presents the detailed elements of an UI event we capture.

To add UI event logging to existing projects with minimum intervention, a compile-time Aspect-Oriented Programming (AOP) technique is used to avoid modifying developers' source code. To add instrumented behaviors (e.g., logging) to specific execution points in a program, AOP "weaves" extra logic at compilation time. Similar instruments have been used for different purposes, such as providing on-device deep links [2]. Android applications, typically written in Java, are commonly built by the Gradle system that compiles Java source code into bytecode for execution. Therefore, we developed a Gradle plugin, *Doppio Transformer*, that can be included in a developer's Android project. It defines the logging logic for a set of interface methods defined by the Android SDK (see Table 2, which

Information	**Execution Example** (Corresponding to Figure 3d)		
Class name	`SignInFragment`		
Method name	`onItemClick`		
List of arguments and their runtime values	**Type**	**Name**	**Runtime Value**
	`AdaptivеView`	`parent`	`android.widget.GridView` - Absolute position: `0,530-1080,1734` - Package:type/entry: `app:id/avatars`
	`View`	`view`	`com.google.samples.apps.topeka.widget.AvatarView` - Absolute position: `0,730-1080,1154` - Package:type/entry: `app:id/avatar`
	`int`	`position`	9
	`long`	`id`	9
Timestamps	Start time T^{start} and finishing time T^{end} of the method execution at the system time.		

Table 1. Key information that Doppio captures for every run-time execution of target input method.

Class	**Method**	**UI Components**
`View`	`onClick, onDrag, onLongClick`	View elements, e.g., a `Button`, `Image`, `TabItem`, and `TextView`.
`AdapterView`	`onItemClick, onItemLongClick, onItemSelected`	Items of a `ListView` or `Spinner`.
`ActionMenuView`	`onMenuItemClick`	Items of a menu (e.g., a Toolbar).
`AppCompatActivity`	`onMenuOpened, onMenuItemSelected`	Items of a menu.
`Activity`	`onBackPressed, onOptionsItemSelected`	The back button; items of an options menu.

Table 2. UI-related methods that Doppio automatically intercepts at compilation time.

can be expanded easily). The Doppio Transformer is built based on the Gradle Transform API and Javassist (Java Programming Assistant), a library that simplifies bytecode weaving [11, 12]. Intercepted UI events are logged using the standard utility logger to minimize the runtime overhead on devices, which will be parsed by the Log Parser of our IDE plugin via Android Debug Bridge (ADB).

Screencast Video Segmentation

While Doppio Transformer provides useful information of *when* a method execution starts and ends, a remaining challenge is to identify the *exact* time when UI elements are actually rendered on the screen. Recent UI design trend for mobile applications, including Google's Material Design [23], highly adopts motion. "Motion design" uses animation to provide a smooth transition between UI changes, such as adding visual elements or switching between views. To help developers observe the method behavior, Doppio aims to capture the entire animation in response to a user action.

To detect the states of UI rendering, one approach is to grab UI hierarchy continuously and observe if elements are rendered, similar to how prior work parses mobile interfaces [13, 14]. However, our goal is to provide a lightweight tool with minimum runtime overheads. Therefore, we adopt an approach similar to MixT [9], which segments the screencast video based on timestamps and screen activities after an interaction flow is captured.

We assume that user interaction is initiated at least 0.3 seconds after animation finishes. Given a screencast video and a set of method executions $\{E_1, E_2, \ldots E_n\}$ in an interaction flow, where each E_i starts at time T_i^{start} and ends with time T_i^{end} (mapped to the video timeline), we adjust T_i^{end} by examining the frame differences until the next method start time T_{i+1}^{start} or the end of the video. For video frames $\{F_1, F_2, \ldots F_m\}$ between $[T_i^{end}, T_{i+1}^{start}]$, we compute the pixel difference δ_j between consecutive frames F_j and F_{j+1} in grayscale. When a sequence of differences $\{\delta_j, \delta_{j+1}, \ldots \delta_k\}$ within 0.3 seconds is all smaller than a threshold (0.1% of screen change), we set the end time T_i^{end} to be the video time of frame F_k. We repeat this process until all the segments are adjusted. Our implementation processes videos in the MP4 format (recorded via ADB) using OpenCV[3] in Java. The length of a demonstration is limited to the device storage when it is tested on device.

Method Identification and Matching

After acquiring the information of method executions and a segmented screencast video, our goal is to identify the corresponding source code of developers' project. As an IDE plugin, Doppio accesses the detailed code structure beyond plain text of the code. This is achieved using Program Structure Interface (PSI) [33] provided by IntelliJ IDEA[4], an open source project that Android Studio IDE is based on. PSI provides a powerful API to locate a program element in a hierarchy, including a class, method, variable, source code, and comment.

Our plugin traverses the codebase and identifies all the methods shown in Table 2. In the Android framework, callbacks can be declared in multiple ways, and a declared class may include more than one method of the same name embedded in anonymous classes. Doppio is capable of finding common types of method declaration, including:

- An override method declared in a class that extends UI classes like `FragmentActivity`, annotated by the Javadoc tag `@Override`.
- An anonymous callback assigned to a variable in a class or a class' method, such as the example on Page 5.
- An anonymous callback that is directly assigned to a `View` object, such as the code below:

```
mAdapter.setOnItemClickListener(
    new CategoryAdapter.OnItemClickListener() {
      @Override
      public void onClick(View v, int position) {
        // Executes business logic for a click event.
      }});
```

For each method execution E_i that contains the logged names of the class and method (see Table 1), Doppio matches the method element in user's project through a recursive process in every declared class. Via PSI, it then retrieves the method's code and automatically creates or modifies the method's comment block to include a unique Doppio link in the form of `/** @look doppio/UNIQUE_ID */`. This link can trigger state highlights in the Viewer via the right-click menu in the IDE. This practice adopts the Javadoc format that developers are used to [55]. In addition, Doppio keeps a record of method's code for each capture and identifies the changed lines between versions by code structure comparison.

Interactive Viewer

Finally, Doppio presents the processed execution results of method behaviors in a Web-based Viewer that can be interactively reviewed and shared with coworkers or online.

Diagram Layout. Our viewer visualizes the flow of method executions in a screenflow diagram organized by classes. Figure 5 shows the layout rendering mechanism, where 1) each column represents a unique class with one or more methods, and 2) the flow progresses along the axes. Doppio starts from the first method execution E_i, rendering it in the first available grid at $(column_i, row_i)$ where i=1. For the following execution E_{i+1}, if it comes from the same class with E_i, place in the same column but the next row $(column_i, row_{i+1})$; otherwise, move to the adjacent column $(column_{i+1}, row_i)$ for a new class, or the next row in its class column $(column_i, row_{i+1})$.

Our layout algorithm might not be screen real-estate efficient as the resulted diagram can be sparse especially when consecutive executions span multiple classes horizontally or one class vertically. However, it provides potential benefits. First, it helps developers focus on the project structure. The columns imply functionality grouping derived by class design based on software engineering practices. Second, if the same method is executed multiple times, a column presents all the examples that can be compared visually, similar to a design gallery [29].

Execution Review. Once obtained the metadata, each execution presents the screenshot captured *before* the interaction, with the target View *region* highlighted and an arrow pointing to the next execution (see Figure 3c-d). We chose to visualize the UI region instead of a specific touch point since regions are often what developers are interested in when designing app layouts. If the region is not passed by the arguments (e.g., by `onBackPressed` or customized methods), arrows will be pointed from the screen border.

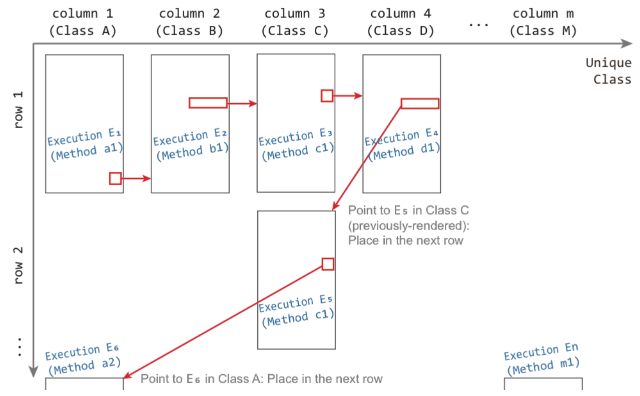


Figure 5. Mechanism how Doppio layouts a class-based screenflow diagram based on method execution sequence.

[3] Open Source Computer Vision Library. http://opencv.org/
[4] https://www.jetbrains.com/idea/

In the IDE, any line with a Doppio link will be decorated with an icon, and developers can right-click on "*View Execution*" from the menu (Figure 3f) to highlight the method in the Viewer. The Doppio IDE plugin runs a local Web server to host the Viewer that renders the screenflow diagram, replays videos dynamically, and handles user interaction using HTML5, D3.js, and jQuery.

RESULTS

To examine the generality of Doppio and the quality of the captured documentation, we conducted an experiment (i.e., *Pre-Study*) using Doppio's automatic approach. We gathered 10 open source repositories from different authors by looking for Android projects on GitHub. The projects were selected from the search results sorted by popularity based on the number of stars. We also specified projects from the samples that Google provides [20]. We filtered out projects that have few or no interactive components, such as a test tool. The final list of testing apps contains a variety of UI elements, including lists, menu bars, tabs, cards, buttons, sliders, checkboxes, text fields, and popup dialogs (see Appendix I). Motion design was seen in 8 of the 10 apps.

Test Environment. We ran applications with an Android Studio 2.3 IDE with Doppio. The software was running on a Mac Pro desktop machine with 32 GB memory and a 3.5GHz 6-Core processor. Each app was running on a Pixel smartphone with 4 GB memory and 32 GB storage running Android 7.1.2, connected to the desktop via USB. Screencast videos were recorded as 1080x1920 pixels from the 5-inch display of the phone.

Methods and Results. For each application, we defined an interaction flow using Espresso by demonstration. We measured the processing time and the accuracy for capturing each method's execution. In particular, we examined how well the video segment covered the method behavior. We repeated the test for each app three times for acquiring aggregated statistics.

Table 3 shows the performance of Doppio. Each row presents the average result from 3 tests of an app. Out of the 90 total clicks in the flows we tested, 82% (74 events) were handled by 26 unique callbacks, where 22 handlers were automatically identified by Doppio and 4 were customized methods from 3 apps that we had to manually annotate. Doppio successfully captured all the execution information of these events and their UI responses. Handlers for other operations that were not specified in the source projects, such as text input, were not traced by our tool. On average, raw videos of a 36.85-second length are extracted to present active UI changes of 16.17 seconds in total.

Figure 6 presents selected results from these applications (with the arrows removed to preserve space). Doppio effectively identifies useful segments that demonstrate user interactions. Doppio precisely highlights UI views to present the moments when user interactions were initiated and how UIs responded for a method. For example, App#5 contains multiple View Fragments and buttons. The captured screenshots are effective in showing the UI progress (see Figure 6a). For App#4 that heavily uses animated transitions between views, Doppio accurately identifies the ends of animations such that the video snippets show the exact transitions (see Figure 6b). It also contains multiple class extensions, where Doppio effectively points out how elements on the same screen are handled by different classes (see Figure 3c). In App#3, #7, and #9 that all contained a menu, Doppio illustrates how each menu item is mapped to a different View—these menu items are handled by the same method but with different runtime argument values (e.g., `position` of the clicked item in a list). Appendix II presents more example results from the 10 apps we tested; Appendix III shows an example with a long input sequence of 25 input actions that span 8 classes and 10 methods in a 2-minute long demonstration.

App	Raw video length (sec)	Total clip length (sec)	Process time (sec)	# of video frames	# of input actions tested	# of input methods executed	Start time of video (%)*	End time of video (%)*
#1	55.94	38.25	16.57	1769.3	18	18 (100%)	0 / 100 / 0	0 / 100 / 0
#2	20.55	14.27	9.02	961.3	5	5 (100%)	0 / 100 / 0	0 / 100 / 0
#3	20.23	7.18	4.17	352.3	7	4 (100%)	0 / 100 / 0	0 / 100 / 0
#4	51.75	26.11	11.34	1295.7	20	9 (100%)	0 / 100 / 0	0 / 100 / 0
#5	32.09	15.86	68.3	727.3	8	8 (100%)	0 / 100 / 0	0 / 100 / 0
#6	27.07	13.67	7.27	616.7	6	6 (100%)	0 / 100 / 0	0 / 100 / 0
#7	47.97	16.84	12.45	1460.7	6	6 (100%)	5.5 / 94.5 / 0	0 / 100 / 0
#8	46.97	8.29	12.66	1363.5	7	7 (100%)	0 / 100 / 0	0 / 100 / 0
#9	31.81	8.78	5.78	621.3	5	5 (100%)	0 / 100 / 0	0 / 100 / 0
#10	34.15	12.56	8.59	881.7	8	6 (100%)	0 / 100 / 0	0 / 100 / 0
AVE	36.85	16.17	9.47	1005.0	9.0	7.4 (100%)	0.55 / 99.45 / 0	0 / 100 / 0

Table 3. Doppio's captures from 10 Android apps.
**** Screenshots taken too early / at exact timing / too late.***

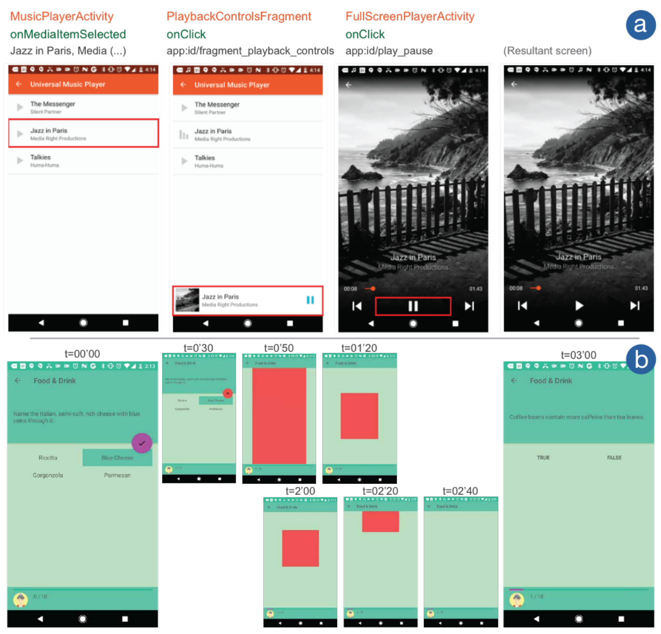


Figure 6. (a) The classes, methods, key argument values, and screenshots captured by Doppio for App#5 (Music Player). (b) A series of intermediate frames from a video segment for App#4 (Topeka) that uses motion design. Video is effective in showing the exact transition between the start and end screens.

USER EVALUATION

We evaluated Doppio's usability via two laboratory studies using the materials generated by Doppio with 16 participants. The first study with 8 engineers focused on how the method-specific visual examples support *code understanding*. Instead of showing the entire screenflow diagram, we specified a set of methods and presented *only* their associated visual examples. The second study with another 8 engineers tested if the class-based screenflow diagrams assist *code finding* and gathered users' feedback on Doppio's code *change tracking* capability.

In these studies, we selected three from the 10 open source projects used in the Pre-Study, including App#3 "*Navigation Drawer*" as a warmup and App#5 "*Music Player*" and App#4 "*Topeka*" for the main tasks. The main projects were chosen because their features are easy to understand, but the code has a moderate degree of complexity and includes a variety of methods that handle user input. We slightly modified the test scripts to simplify the interaction flows for both apps. We used the same smartphone and desktop computer that we used for the Pre-Study. The computer was connected to a dual 24-inch LCD displays, each with a 1920x1200 pixel resolution.

We compared Doppio with a Baseline that provided a standard Android Studio IDE with counterbalanced order of conditions. Participants were selected via an internal invitation and all had experiences using Android Studio for app development. None of them had seen the projects used in the studies. Each participant was compensated with a $25 gift card for their participation in a one-hour session.

Study 1: Effectiveness for Code Understanding

Building upon prior studies that suggested the benefits of visual or video materials for learning unfamiliar concepts [48, 9], we hypothesized that programmers can understand a method's behavior of an unfamiliar project more accurately with the visual examples that Doppio captured than existing practice in constrained time. We conducted an informal within-subject study with 8 software engineers (1 female), aged 25 to 42 years (Mean=32.5) from an IT company. In the Doppio condition, we only showed the captured information of the target methods, while each visually presented only the screenshots, video clip, and runtime argument values.

Each session started with the warm-up project to help participants familiarize with the environment. Then, we presented App#5 by walking through the same interaction flow from the test script we used in the Pre-Study. We asked participants to describe the behaviors of three specified methods in text in 15 minutes as if they were documenting source code for their everyday work projects (Task 1). Documentation of parameters and return values was not required. We then introduced App#4 and the other condition and asked to write for another three methods (Task 2). Details about the selected methods and their behaviors are listed in Appendix IV.

Results. All participants but one completed both tasks. One user failed to initiate writing for Method 3 in the Baseline condition under the time constraint. We found that participants constantly replayed the videos to observe the UI changes with Doppio. On average, videos were replayed 33.3 times (*SD*=31.4) for Task 1 and 27.8 times (*SD*=14.6) for Task 2. Without Doppio, participants mainly focused on code reading and tracing; only two users reran the apps.

Of the 47 completed descriptions from the two tasks, the average length was 19 words (SD=13.4) for the Baseline and 22 words (SD=23.7) for Doppio. While the average word length in the Doppio condition was slightly longer with larger variation, participants made more concise descriptions when we looked into how a method's behavior was documented by its content. Specifically, descriptions from the Baseline mainly were verbosely translated from code logic, line by line. For example, for Method 1 in Task 1, 75% of the Baseline descriptions listed all the three detailed `if-else` conditions one by one (e.g., "*Either plays the selected media (with the given ID) when it's playable or browses the selected media when it's browsable. Throws an exception if the item is not in both cases*." by P5*),* where the descriptions using Doppio's visual examples were less verbose (e.g., "*Plays the media item if playable or navigates to the selected item*." by P2). We also noted that with Doppio, descriptions included the concrete visuals more, such as "*disappear*", "*grid*", "*pop up*", "*bigger*", "*inside*", "*transition*", "*animation*" for Task 2 that applied motion design.

Doppio helped participants correct errors. For example, P8's strategy was to first comprehend through reading source code and then verify via the visual examples. For the method that shows the full-screen view of the user-selected song via "`navigateToBrowser(item.getMediaId())`" in Task 1, P8 originally wrote "*opens it in a browser*". After seeing the video, he immediately changed to "*opens it for browsing*" as the method name did not infer an actual browser but an abstraction. P8 pointed out the difference as "*When looking at completely unfamiliar code, I had assumptions on what some of the method are doing based on their names. Being able to see what happens in the UI allowed me to be more confident in my assumptions, and saved me some digging*."

Participants found the screenshots useful in understanding the methods' behaviors (Median ratings=4.5 out of the 5-point Likert-scale), so is the videos (Median=4.5), similar to MixT's results of mixed-media tutorials [9]. P3 explained, "*In the context of debugging, it was extremely helpful to see activities/fragments transitions to validate the code logic*." P4 commented, "*Nothing is more intuitive than seeing the effects of a method the way a user perceives it*." These results suggest that Doppio helped developers understand and verify code behaviors via visual examples.

Study 2a: Effectiveness for Code Finding

After verifying how Doppio's segmented information supported developers in understanding methods' behaviors, we aimed to test if it would be useful to present the method overview of an interaction flow while preserving the visual demonstrations. We conducted another within-subject study with 8 software engineers different from Study 1 (all males, aged 22 to 39 years, Mean=28.25), from the same IT company. The first part of Study 2 hypothesized that programmers can efficiently find methods of interest in an unfamiliar project with Doppio than existing practice. In the Doppio condition, we showed the Viewer as Figure 3c presents. The entire study took 44 minutes on average, including a 3-minute introduction to Doppio's Viewer.

Similar to Study 1, each session included a warm-up and the same two Android projects and interaction flows. However, we assume that method finding can be more challenging than Study 1 given that App#5 includes 42 classes and 459 declared methods in 5 packages and 1 sub-package, and App#4 has 57 classes and 672 declared methods in 7 packages and 4 sub-packages (see Table 4). Therefore, for each project, we limited the tasks to find two from the three methods of each project used in Study 1:

- Task 1 asked participants to find the methods given two specific interactions and screens in App#5 ("*Please find the method that responds to song selection*" and "(…) *handles play/pause of a song in the song full-screen view*") in 10 minutes.
- Task 2 asked to find the methods that "*responds to the question type that user selects*" and "*handles and records the answer submission*" in App#4.

Results. All participants but one completed both the tasks. One user failed to find one method in App#5 with the Baseline condition. On average, participants spent 3.75 and 5.38 minutes with Doppio (all completed the tasks early), which saved 55% and 23% of time from the Baseline where participants spent 8.25 and 7 minutes for each task (while three of eight participants used up their 10-minute limit). In terms of performance comparing with self, each participant completed the task twice faster with Doppio than without it. All participants thought that it was faster to find target methods with Doppio (Median ratings=5). We observed that their strategy was to use Doppio to locate methods in the projects and trace code to verify the behaviors, whereas without Doppio, participants either relied on code search by keywords (e.g., "pause" or "click" in Task 1 or "submit" in Task 2) or traverse each class via breath-first search.

Correctness of method finding drew a difference between the two conditions (see Table 5). In Task 1, using Doppio, both of the user-identified methods were 100% correct; with the Baseline, each method was 75% correct, which included an incomplete answer and an incorrect method in the correct class respectively. Task 2 had a major drop (see Appendix V for detailed code snippets). Using Doppio, the first method got 100% correctness; with the Baseline, one participant was confused by a similar class–he chose the class `CategorySelectionActivity` that handles the app's main control for account settings instead of the detailed `CategorySelectionFragment` that presents and handles the UI grids of quiz types). The second method was very challenging because of the app design, which sets the input callback dynamically via a base class. The behavior would require time and programming experience to trace the source code. None of the Baseline participants got this method correctly; one participant using Doppio also gave the same incorrect answer, who later explained how he got confused and couldn't figure out under the time constraint.

Task	States (clicks)	Invoked classes	Total classes in project	Invoked callbacks	Total methods in project
1 (App#5)	6	4	42	4	459
2 (App#4)	9	5	57	7	672

Table 4. Number of interactions in test scripts and their invoked classes and methods compared to project methods.

Task-Method	Time (minutes)		Correctness		Error type
	Baseline	Doppio	Baseline	Doppio	
1-1	8.25	3.75	75%	100%	Incomplete
1-2			75%	100%	Incorrect method
2-1	7	5.38	75%	100%	Incorrect class
2-2			0%	75%	Incorrect class

Table 5. Task performance in Study 2a.

Similar to Study 1, participants found the screenshots useful in understanding the methods' behaviors (Median=5), so is the videos (Median=4.5). They spent 17.5 seconds (SD=5.4) in total reviewing detailed snippets for Task 1 and 32.4 seconds (SD=9.5) for Task 2. Video clips, mostly 2-3 seconds, were reviewed 10.75 times (SD=6.85) for 5 clips in Task 1 and 7.75 times (SD=1.5) for all 9 clips in Task 2.

Study 2b: Support of Code Change Understanding

Finally, we investigated how Doppio's code tracking capability supports developers in understanding changes. Immediately after each task in Study 2a, we asked participants to modify the app behavior by adding a confirmation dialog (after tapping a song View in App#5, show "*Starts playing this song?*"; before leaving the quiz in App#4, show "*Leaves the quiz?*"). Participants were asked to modify the code to call a declared method we provided that handles the new behavior and rerun the test script. Then, in the Doppio condition, we presented and explained the updated screenflow diagram. We asked participants of both conditions to write a revision description as if they were submitting this code change to a coworker. Appendix VI and VII present the diagrams that participants experienced after code revision.

Results. Six of the eight participants chose to include a screenshot or a demo video linked from the revision in their descriptions. Another participant asked if he could include the link to the Viewer page although he did not include any link in the first task with the Baseline. P1 explained, "*I definitely would have added screenshots/links to Doppio—*

it would have been very useful to include." All participants thought that it was faster to track changes with Doppio (Median=4.5). When describing Doppio's advantages, P3 said "*provide more visibility of recent changes, share with other team peers to know what your CL change, and better for team demo/presentation*." This indicates that Doppio's automatic approach of app state identification is useful and supports our motivation of providing visual examples as software documentation for sharing.

DISCUSSION AND OPPORTUNITIES

Overall, we received very positive feedback from participants on their experience using Doppio. All participants found it easy to use our tool (Median=5) and easy to review UI behaviors of an app with Doppio (Median =5). All rated 5 that the screenflow diagrams were useful, and the concept of visualizing app behaviors in an interactive viewer was straightforward. In Study 2, P1 commented, "*I felt confident about my understanding of the event handling much quicker (as in, almost immediately) with Doppio, whereas without Doppio I had to do a lot of looking around*." Other participants commented that Doppio is "*a very powerful tool in terms of understanding the app itself and the code base*" (P2), "*very handy for developers beginning with a project*" (P4), and "*great for bug reproduction and tracing with video recorded*" (P7). Overall, we were pleased to find that all the participants preferred to develop apps with Doppio over without its support (Median=5) and all strongly agreed that they would want to have this plugin if it is available.

Doppio's approach is based on the access to the source code. Therefore, support of cross-app interaction (e.g., switching to other app intents for operations) is limited if such a behavior is not part of the source project. Below we describe more of Doppio's limitations and opportunities.

Interactive Debugging. From our observations and user feedback, we strongly believe that Doppio's automatic approach of visualizing input callbacks can be powerful in developing an interactive system. We demonstrated how Doppio can effectively assist developers in code understanding, tracing, and testing with the visual examples. But we also acknowledge the opportunities to support interactive debugging that we have not yet shown. P2 and P6 suggested having the tool better integration with the IDE, such as automatically inspecting methods, pausing at the target callbacks, and handling obsolete metadata.

Diverse Mobile Input and Output. Our design focuses on specific user inputs of single touchscreen events (such as `onClick`) and visible on-screen user feedback. Our video-based heuristics by comparing frame differences are built upon design principles for such type of interaction. Doppio does not fully handle continuous dynamic input (e.g., sensors, gestures, or speech that involves a diverse collection of event triggers) or looping animation. To support applications of more interactive techniques and feedback, potential solutions include: programming by demonstration for developers to specify a clear link between continuous events and code, and pixel-based UI recognition [16] to identify visual pattern or repetition.

Logic Understanding and Automation. Our current approach does not interpret the logic inside a method. For a function that handles several conditions, we rely on a test session that covers the code. As Study1-P3 pointed out, "*if the logic is fairly complicated with many different paths, it would be useful to see the exact code path (e.g. line number, along with the details of the intent)*." Since Doppio has the access to the source code, it is possible to trace these conditions and visualize them similar to Whyline [38]. Similarly, changed behaviors would not be reflected if not covered by the test script or in demonstration, and different traces and configurations over time are not compared. We seek to bring automation to the process to leverage the existing testing frameworks as several participants suggested (Study1-P2, P5, and P8). P5 in Study 2 commented, "*It would be nice if there were some kind of warning indicator saying they are not being tested if you didn't add tests*."

Platform Integrations. For software development, an important role of documentation is to help other developers to review and modify the code. From Study 2b, Doppio's code and execution tracing could support app iteration on visual design and UI behaviors. Our Web-based Viewer makes it promising to share the materials online for collaboration. For better integration with existing online platforms, we are developing a Chrome plugin that presents the video snippets for code-sharing sites like GitHub. Another promising approach is to export video segments as GIFs that can be embedded in a markup file or webpage.

Example-Based Code Search and Debug. We are excited about the future when a community captures and shares execution results using Doppio. As a crowd-powered IDE has shown to be useful in programing [17], code search based on input-driven examples can be introduced when links between source code and runtime examples are available. By presenting the execution results and code snippets, developers can visually compare, reason, and program interactively. Last but not least, to support app search, we also look forward to further generating the method summary in natural language or patterns learned from the captured information similar to prior work for code summary [49] and code changes [45, 5, 52].

CONCLUSION

We present Doppio, a tool that automatically tracks and visualizes UI flows and their changes based on source code elements and their revisions. We integrate Doppio, as an IDE plugin, seamlessly into a development workflow to generate interactive screenflow diagrams organized by the callback methods and input sequences. We tested Doppio on a range of open source projects, which present compelling results on visual documentation. We also evaluated Doppio with 16 professional developers and gained positive feedback.

REFERENCES

1. Dimitar Asenov, Otmar Hilliges, and Peter Müller. 2016. The Effect of Richer Visualizations on Code Comprehension. In Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16). ACM, New York, NY, USA, 5040-5045. DOI: https://doi.org/10.1145/2858036.2858372
2. Tanzirul Azim, Oriana Riva, and Suman Nath. 2016. uLink: Enabling User-Defined Deep Linking to App Content. In Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '16). ACM, New York, NY, USA, 305-318. DOI: http://dx.doi.org/10.1145/2906388.2906416
3. Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. 2013. Interactive record/replay for web application debugging. In Proceedings of the 26th annual ACM symposium on User interface software and technology (UIST '13). ACM, New York, NY, USA, 473-484. DOI: http://dx.doi.org/10.1145/2501988.2502050
4. Brian Burg, Andrew J. Ko, and Michael D. Ernst. 2015. Explaining Visual Changes in Web Interfaces. In Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15). ACM, New York, NY, USA, 259-268. DOI: https://doi.org/10.1145/2807442.2807473
5. Raymond P.L. Buse and Westley R. Weimer. 2010. Automatically documenting program changes. In Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE '10). ACM, New York, NY, USA, 33-42. DOI=http://dx.doi.org/10.1145/1858996.1859005
6. Kerry Shih-Ping Chang and Brad A Myers. 2012. WebCrystal: understanding and reusing examples in web authoring. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, 3205–3214.
7. Hsiang-Ting Chen, Tovi Grossman, Li-Yi Wei, Ryan M. Schmidt, Björn Hartmann, George Fitzmaurice, and Maneesh Agrawala. 2014. History assisted view authoring for 3D models. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14). ACM, New York, NY, USA, 2027-2036. DOI=http://dx.doi.org/10.1145/2556288.2557009
8. Yan Chen, Sang Won Lee, Yin Xie, YiWei Yang, Walter S. Lasecki, and Steve Oney. 2017. Codeon: On-Demand Software Development Assistance. In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17). ACM, New York, NY, USA, 6220-6231. DOI: https://doi.org/10.1145/3025453.3025972
9. Pei-Yu (Peggy) Chi, Sally Ahn, Amanda Ren, Mira Dontcheva, Wilmot Li, and Björn Hartmann. 2012. MixT: automatic generation of step-by-step mixed media tutorials. In Proceedings of the 25th annual ACM symposium on User interface software and technology (UIST '12). ACM, New York, NY, USA, 93-102. DOI: http://dx.doi.org/10.1145/2380116.2380130
10. Pei-Yu (Peggy) Chi, Yang Li, and Björn Hartmann. 2016. Enhancing Cross-Device Interaction Scripting with Interactive Illustrations. In Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16). ACM, New York, NY, USA, 5482-5493. DOI: https://doi.org/10.1145/2858036.2858382
11. Shigeru Chiba. 2000. Load-Time Structural Reflection in Java. In ECOOP 2000: Object-Oriented Programming. Springer.
12. Shigeru Chiba and Muga Nishizawa. 2003. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In GPCE 2003: Generative Programming and Component Engineering. Springer.
13. Biplab Deka, Zifeng Huang, and Ranjitha Kumar. 2016. ERICA: Interaction Mining Mobile Apps. In Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16). ACM, New York, NY, USA, 767-776. DOI: https://doi.org/10.1145/2984511.2984581.
14. Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In Proceedings of the 30th Annual Symposium on User Interface Software and Technology (UIST '17). ACM, New York, NY, USA.
15. Stephan Diehl. 2007. Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
16. Morgan Dixon and James Fogarty. 2010. Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10). ACM, New York, NY, USA, 1525-1534. DOI: https://doi.org/10.1145/1753326.1753554
17. Ethan Fast, Daniel Steffee, Lucy Wang, Joel R. Brandt, and Michael S. Bernstein. 2014. Emergent, crowd-scale programming practice in the IDE. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14). ACM, New York, NY, USA, 2491-2500. DOI=http://dx.doi.org/10.1145/2556288.2556998
18. Shiry Ginosar, Luis Fernando De Pombo, Maneesh Agrawala, and Bjorn Hartmann. 2013. Authoring multi-stage code examples with editable code histories. In Proceedings of the 26th annual ACM symposium on User interface software and technology (UIST '13). ACM, New York, NY, USA, 485-494. DOI: http://dx.doi.org/10.1145/2501988.2502053
19. Google Inc. Android Studio. https://developer.android.com/studio/ (accessed: 09/01/2017).
20. Google Inc. Google Samples. https://github.com/googlesamples/ (accessed: 09/01/2017).
21. Google Inc. Testing UI for a Single App. https://developer.android.com/training/testing/ui-testing/espresso-testing.html (accessed: 09/01/2017).
22. Google Inc. Topeka for Android. https://github.com/googlesamples/android-topeka/tree/java (accessed: 09/01/2017).
23. Google Inc. Material Design. https://material.io/ (accessed: 09/01/2017).
24. Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. 2009. Generating photo manipulation tutorials by demonstration. ACM Trans. Graph.

28, 3, Article 66 (July 2009), 9 pages. DOI=http://dx.doi.org/10.1145/1531326.1531372

25. Tovi Grossman and George Fitzmaurice. 2010. ToolClips: an investigation of contextual video assistance for functionality understanding. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10). ACM, New York, NY, USA, 1515-1524. DOI=http://dx.doi.org/10.1145/1753326.1753552

26. Tovi Grossman, Justin Matejka, and George Fitzmaurice. 2010. Chronicle: capture, exploration, and playback of document workflow histories. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology* (UIST '10). ACM, New York, NY, USA, 143-152. DOI=http://dx.doi.org/10.1145/1866029.1866054

27. Yiyang Hao, Ge Li, Lili Mou, Lu Zhang, and Zhi Jin. 2013. MCT: a tool for commenting programs by multimedia comments. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13). IEEE Press, Piscataway, NJ, USA, 1339-1342.

28. Dorsaf Haouari, Houari Sahraoui, and Philippe Langlais. 2011. How Good is Your Comment? A Study of Comments in Java Programs. 2011 International Symposium on Empirical Software Engineering and Measurement, Banff, AB, Canada, 137-146. DOI=10.1109/ESEM.2011.22

29. Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. 2008. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In Proceedings of the 21st annual ACM symposium on User interface software and technology (UIST '08). ACM, New York, NY, USA, 91-100. DOI: https://doi.org/10.1145/1449715.1449732

30. Björn Hartmann, Sean Follmer, Antonio Ricciardi, Timothy Cardenas, and Scott R. Klemmer. 2010. d.note: revising user interfaces through change tracking, annotations, and alternatives. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10). ACM, New York, NY, USA, 493-502. DOI: https://doi.org/10.1145/1753326.1753400

31. Joshua Hibschman and Haoqi Zhang. 2015. Unravel: Rapid Web Application Reverse Engineering via Interaction Recording, Source Tracing, and Library Detection. In Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15). ACM, New York, NY, USA, 270-279. DOI: https://doi.org/10.1145/2807442.2807468.

32. Joshua Hibschman and Haoqi Zhang. 2016. Telescope: Fine-Tuned Discovery of Interactive Web UI Feature Implementation. In Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16). ACM, New York, NY, USA, 233-245. DOI: https://doi.org/10.1145/2984511.2984570

33. JetBrains. Program Structure Interface. IntelliJ Platform SDK DevGuide. http://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi_files.html (accessed: 03/10/2017).

34. Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17). ACM, New York, NY, USA, 737-745. DOI: https://doi.org/10.1145/3126594.3126632

35. Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan Borchers. 2011. Stacksplorer: call graph navigation helps increasing code maintenance efficiency. In Proc. UIST '11, 217-224.

36. Jun Kato, Sean McDirmid, and Xiang Cao. 2012. DejaVu: integrated support for developing interactive camera-based programs. In Proceedings of the 25th annual ACM symposium on User interface software and technology (UIST '12). ACM, New York, NY, USA, 189-196. DOI: https://doi.org/10.1145/2380116.2380142

37. Jun Kato, Daisuke Sakamoto, and Takeo Igarashi. 2013. Picode: inline photos representing posture data in source code. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13). ACM, New York, NY, USA, 3097-3100. DOI: https://doi.org/10.1145/2470654.2466422

38. Andrew J. Ko and Brad A. Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04). ACM, New York, NY, USA, 151-158. DOI=http://dx.doi.org/10.1145/985692.985712

39. Douglas Kramer. 1999. API documentation from source code comments: a case study of Javadoc. In Proceedings of the 17th annual international conference on Computer documentation (SIGDOC '99). ACM, New York, NY, USA, 147-153. DOI=http://dx.doi.org/10.1145/318372.318577

40. Jan-Peter Krämer, Joel Brandt, and Jan Borchers. 2016. Using Runtime Traces to Improve Documentation and Unit Test Authoring for Dynamic Languages. In Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16). ACM, New York, NY, USA, 3232-3237. DOI: https://doi.org/10.1145/2858036.2858311

41. Sandeep K. Kuttal, Anita Sarma, and Gregg Rothermel. 2014. On the benefits of providing versioning support for end users: An empirical study. ACM Trans. Comput.-Hum. Interact. 21, 2, Article 9 (February 2014), 43 pages. DOI: https://doi.org/10.1145/2560016

42. Timothy C. Lethbridge, Janice Singer, and Andrew Forward. 2003. How Software Engineers Use Documentation: The State of the Practice. IEEE Softw. 20, 6 (November 2003), 35-39. DOI=http://dx.doi.org/10.1109/MS.2003.1241364

43. Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14). ACM, New York, NY, USA, 2481-2490. DOI=http://dx.doi.org/10.1145/2556288.2557409.

44. Henry Lieberman. 1993. Mondrian: a teachable graphical editor. In Watch what I do, Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky (Eds.). MIT Press, Cambridge, MA, USA 341-358.

45. Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. ChangeScribe: a tool for automatically generating commit messages. In Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15), Vol. 2. IEEE Press, Piscataway, NJ, USA, 709-712.

46. Dastyni Loksa, Andrew J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16). ACM, New York, NY, USA, 1449-1461. DOI: https://doi.org/10.1145/2858036.2858252

47. Josip Maras, Jan Carlson, and Ivica Crnkovi. 2012. Extracting client-side web application code. In Proceedings of the 21st international conference on World Wide Web. ACM, 819–828.

48. Richard E Mayer, William Bove, Alexandra Bryman, Rebecca Mars, and Lene Tapangco. "When less is more: Meaningful learning from visual and verbal summaries of science textbook lessons." In: Journal of educational psychology 88.1 (1996), p. 64.

49. Paul W. McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014). ACM, New York, NY, USA, 279-290. DOI=http://dx.doi.org/10.1145/2597008.2597149

50. Hiroaki Mikami, Daisuke Sakamoto, and Takeo Igarashi. 2017. Micro-Versioning Tool to Support Experimentation in Exploratory Programming. In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17). ACM, New York, NY, USA, 6208-6219. DOI: https://doi.org/10.1145/3025453.3025597

51. Brad Myers, Scott E. Hudson, and Randy Pausch. 2000. Past, present, and future of user interface software tools. ACM Trans. Comput.-Hum. Interact. 7, 1, 3-28.

52. Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. 2014. Mining fine-grained code changes to detect unknown change patterns. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, New York, NY, USA, 803-813. DOI: https://doi.org/10.1145/2568225.2568317

53. Stephen Oney and Brad Myers. 2009. FireCrystal: Understanding interactive behaviors in dynamic web pages. In Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on. IEEE, 105–108.

54. Stephen Oney and Joel Brandt. 2012. Codelets: linking interactive documentation and example code in the editor. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12). ACM, New York, NY, USA, 2697-2706. DOI: http://dx.doi.org/10.1145/2207676.2208664Y.

55. Oracle. Javadoc Tool. http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html (accessed: 3/10/ 2017).

56. Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. 2016. MobiPlay: a remote execution based record-and-replay tool for mobile applications. In Proceedings of the 38th International Conference on Software Engineering (ICSE '16). ACM, New York, NY, USA, 571-582. DOI: https://doi.org/10.1145/2884781.2884854.

57. Xin Rong, Shiyan Yan, Stephen Oney, Mira Dontcheva, and Eytan Adar. 2016. CodeMend: Assisting Interactive Programming with Bimodal Embedding. In Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16). ACM, New York, NY, USA, 247-258. DOI: https://doi.org/10.1145/2984511.2984544.

58. Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE '10). ACM, New York, NY, USA, 43-52. DOI=10.1145/1858996.1859006 http://doi.acm.org/10.1145/1858996.1859006

59. Moritz Wittenhagen, Christian Cherek, and Jan Borchers. 2016. Chronicler: Interactive Exploration of Source Code History. In Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16). ACM, New York, NY, USA, 3522-3532. DOI: https://doi.org/10.1145/2858036.2858442

60. Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: using GUI screenshots for search and automation. In Proceedings of the 22nd annual ACM symposium on User interface software and technology (UIST '09). ACM, New York, NY, USA, 183-192. DOI=http://dx.doi.org/10.1145/1622176.1622213

61. YoungSeok Yoon, Brad A. Myers, and Sebon Koo. 2013. Visualization of fine-grained code change history. 2013 IEEE Symposium on Visual Languages and Human Centric Computing, San Jose, CA, USA, 119-126. DOI=10.1109/VLHCC.2013.6645254

APPENDICES

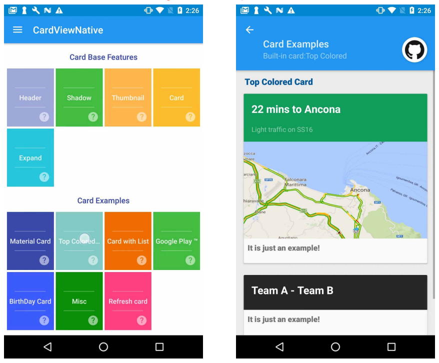
Appendix I. 10 open-sourced Android applications we tested (sorted by author names).

App	Author	App Name	Description (from authors)	Major UI Components	# of screenshots in README (images / GIFs)	GitHub link (https://github.com/)
#1	florent37	Material View Pager	A Material Design ViewPager easy to use library.	A list of four tabs and a navigation drawer.	2 / 8	florent37/MaterialViewPager
#2	Gabriele Mariotti	Card Library	Android Library to build a UI Card.	Multiple grid lists and a navigation drawer. Each grid opens another view of multiple cards.	4 / 0	gabrielemariotti/cardslib
#3	Google	Navigation Drawer	… illustrates a common usage of the DrawerLayout widget in the Android support library.	A main card that opens another view with a navigation drawer with a list of eight options.	0 / 0	googlesamples/android-NavigationDrawer
#4	Google	Topeka	A fun to play quiz that showcases material design on Android.	A grid list that show categories of quizzes. Each quiz includes 10 questions that are answered by checkboxes, sliders, text fields, or buttons.	3 / 0	googlesamples/android-topeka/tree/java
#5	Google	Universal Android Music Player	… shows how to implement an audio media app that works across multiple form factors.	Three lists in order (Main > Genres > Songs) and a sliding card of the song being played, which can be expanded to full screen, with a control bar of three buttons (play/pause, previous, next).	6 / 0	googlesamples/android-UniversalMusicPlayer
#6	Google	unsplash	A window into transitions.	A grid list of various sizes. Each grid opens another full-screen view.	0 / 0	googlesamples/android-unsplash
#7	Google	XYZ Tourist Attractions	… notifies the user when they are in close proximity to notable points of interest.	A list of six cards, each opens a detailed view with a button that launches the Maps app to show a specific location.	1 / 0	googlesamples/android-XYZTouristAttractions
#8	nickbutcher	plaid	… provides design news & inspiration as well as being an example of implementing material design.	A grid list of dynamic sizes. Each grid opens another view that includes buttons for sharing, commenting, or bookmarking.	4 / 1	nickbutcher/plaid
#9	Square	Times Square	Standalone Android widget for picking a single date from a calendar view.	A list of ten tabs, each presents different calendar design. Two pop up the calendar as a dialog.	1 / 0	square/android-times-square
#10	Yalantis	uCrop	Image Cropping Library for Android	A form with buttons, checkboxes, and text fields; Another view for cropping images via buttons and sliders.	0 / 1	Yalantis/uCrop

Appendix II. Example method execution from the 10 applications we tested.

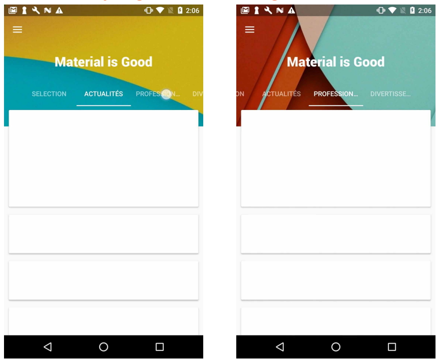
App#1

NativeMenuActivity :: onTopicSelected



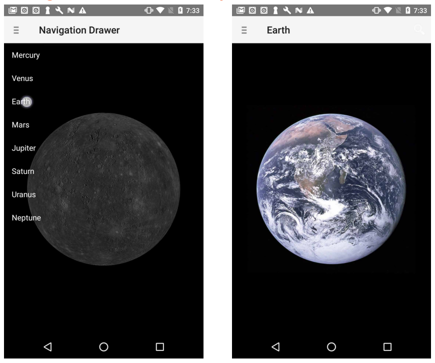
App#2

MainActivity :: getHeaderDesign



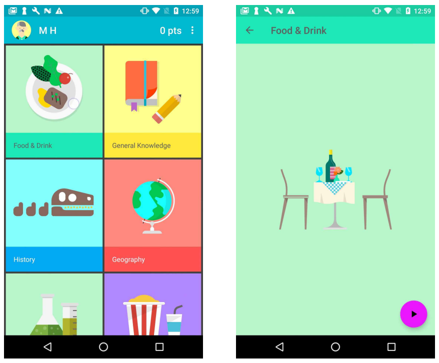
App#3

NavigationDrawerActivity :: onClick



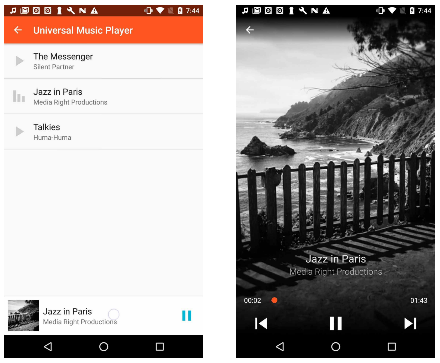
App#4

CategorySelectionFragment :: onClick



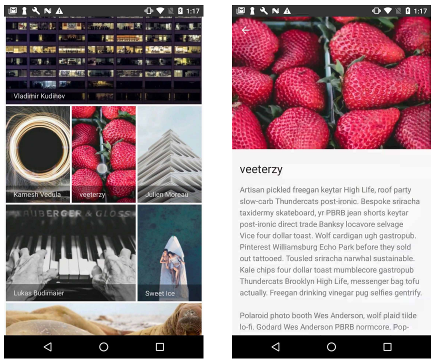
App#5

PlaybackControlsFragment :: onClick



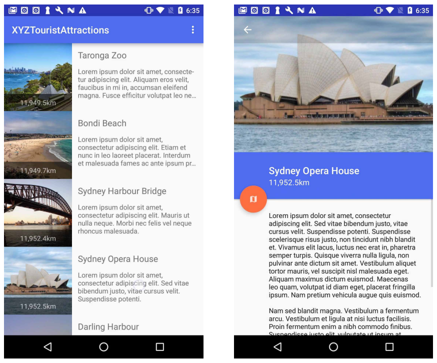
App#6

MainActivity :: onItemSelected



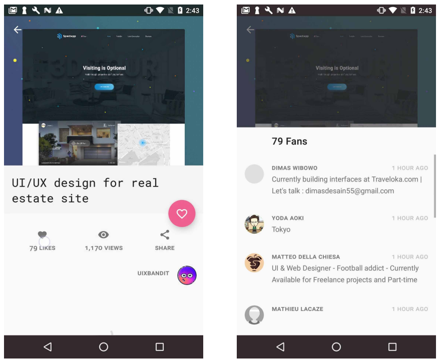
App#7

ViewHolder :: onClick



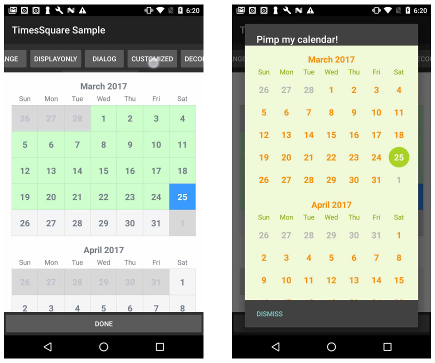
App#8

DribbbleShot :: onClick



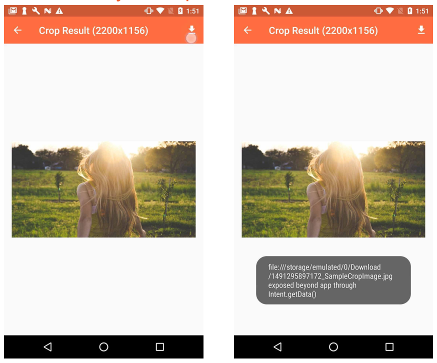
App#9

SampleTimesSquareActivity :: onClick



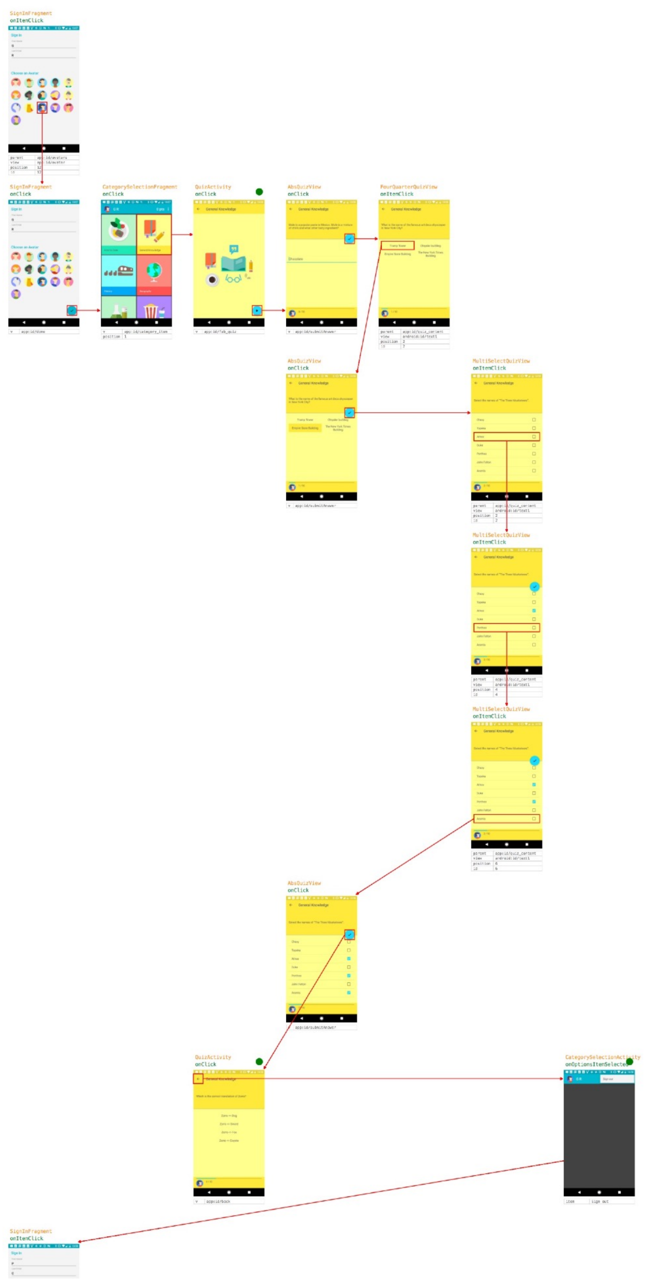
App#10

ResultActivity :: onOptionsItemSelected

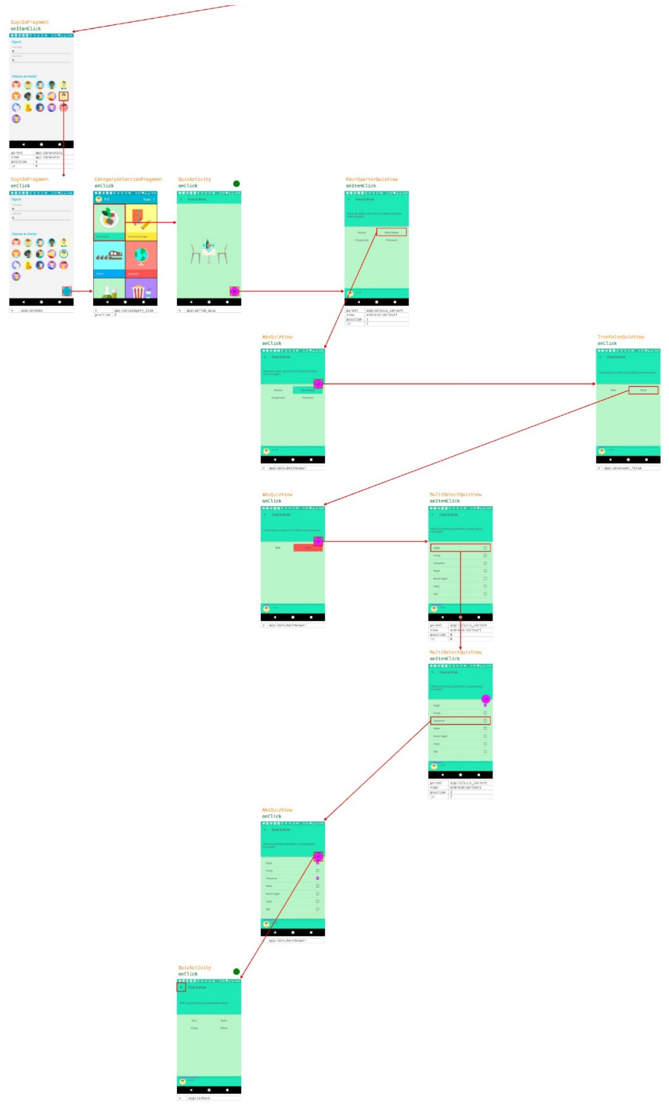

Crop Result (2200x1156)
Crop Result (2200x1156)
file:///storage/emulated/0/Download/1491295897172_SampleCropImage.jpg exposed beyond app through Intent.getData()

Appendix III. Screenflow diagram of App#4 with 25 clicks that involve 8 classes and 10 unique callback methods.

Top page



Bottom page



Appendix IV. Methods we asked participants to describe in Study 1.

Task	App	App Name	Class and Method	Behaviors (Text is not shown to participants)
warmup	#3	Navigation Drawer	`MainActivity :: onItemClick*`	Selects a sample project (shown as a card).
			`NavigationDrawerActivity :: onClick*`	Selects an item from a sliding menu.
1	#5	Universal Android Music Player	`MusicPlayerActivity :: onMediaItemSelected*`	Selects an item of genres/artists/songs from a list.
			`PlaybackControlsFragment :: onClick`	Expands to the full screen to review and control the song being played.
			`FullScreenPlayerActivity :: onClick*`	Handles to play/pause a song.
2	#4	Topeka	`fragment/CategorySelectionFragment :: onClick*`	Selects a category card from a set.
			`activity/QuizActivity :: onClick`	Starts the quiz via a button.
			`widget/quiz/AbsQuizView :: onClick*`	Submits the answer and proceeds to the next question via a button.

* Also used in Study 2.

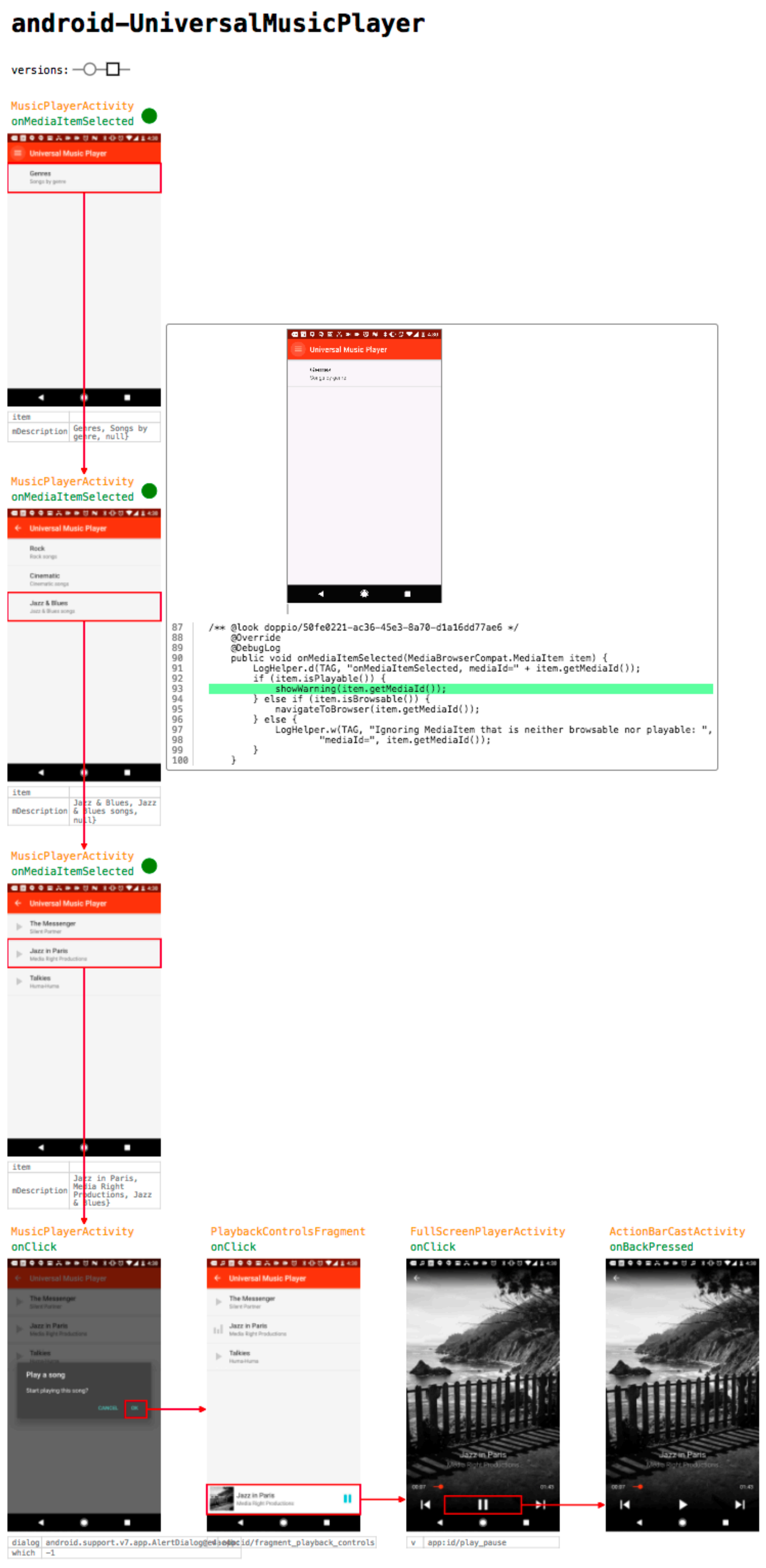
Appendix V. Code snippets of Task 2's methods in Study 2a.

Method	Behavior	Code Snippet from the Correct Class/Method	Incorrect Class/Method Specified by Participants

1	responds to the question type that user selects	In Class `CategorySelectionFragment`:
		
		`mAdapter.setOnItemClickListener(`
		`  new CategoryAdapter.OnItemClickListener() {`
		`    @Override`
		`    public void onClick(View v, int position) {`
		`      Activity activity = getActivity();`
		`      startQuizActivityWithTransition(`
		`        activity,`
		`        v.findViewById(R.id.category_title),`
		`        mAdapter.getItem(position));`
		`    }`
		`  });`
			In Class `CategorySelectionActivity`:
			
			`@Override`
			`public boolean onOptionsItemSelected(`
			`    MenuItem item) {`
			`  switch (item.getItemId()) {`
			`    case R.id.sign_out: {`
			`      signOut();`
			`      return true;`
			`    }`
			`  }`
			`  return super.onOptionsItemSelected(item);`
			`}`

2	handles and records the answer submission	In Class `AbsQuizView`:
		
		The submit button that has the id `submitAnswer` (defined in the layout `answer_submit`) is generated dynamically when a new View is rendered and calls its base class `AbsQuizView`. The button is hidden by default and will be shown dynamically when user enters an answer.
		
		`mSubmitAnswer = (CheckableFab) getLayoutInflater()`
		`        .inflate(R.layout.answer_submit, this, false);`
		`mSubmitAnswer.hide();`
		`mSubmitAnswer.setOnClickListener(`
		`  new OnClickListener() {`
		`    @Override`
		`    public void onClick(View v) {`
		`      submitAnswer(v);`
		`      if (mInputMethodManager.isAcceptingText()) {`
		`        mInputMethodManager`
		`          .hideSoftInputFromWindow(v.getWindowToken(), 0);`
		`      }`
		`      mSubmitAnswer.setEnabled(false);`
		`    }`
		`});`
		
		When clicked, the callback invokes the method that is defined in the same class:
		
		`private void submitAnswer(final View v) {`
		`  final boolean answerCorrect = isAnswerCorrect();`
		`  mQuiz.setSolved(true);`
		`  performScoreAnimation(answerCorrect);`
		`}`
			In Class `QuizActivity`:
			
			A click callback is defined to handle View items such as entering or leaving the quiz, but the submit button with the id `submitAnswer` will be set with this callback in practice:
			
			`private final View.OnClickListener`
			`mOnClickListener = new View.OnClickListener() {`
			`    @Override`
			`    public void onClick(final View v) {`
			`        switch (v.getId()) {`
			`            case R.id.fab_quiz:`
			`                startQuizFromClickOn(v);`
			`                break;`
			`            case R.id.submitAnswer:`
			`                submitAnswer();`
			`                break;`
			`            // ...`
			`        }`
			`    }`
			`};`
			
			This class also defines a method `submitAnswer()` that proceeds to the next question, but it is called via `proceed()`, which is invoked by a method in the class `AbsQuizView` via `((QuizActivity) getContext()).proceed();`:
			
			`public void proceed() {`
			`    submitAnswer();`
			`}`
			
			`private void submitAnswer() {`
			`    mCountingIdlingResource.decrement();`
			`    if (!mQuizFragment.showNextPage()) {`
			`        mQuizFragment.showSummary();`
			`        setResultSolved();`
			`        return;`
			`    }`
			`    setToolbarElevation(false);`
			`}`

Appendix VI. Final screenflow diagram shown in the Viewer for Task 1 after code modification in Study 2b.



android-UniversalMusicPlayer

versions:

MusicPlayerActivity
onMediaItemSelected

item	
mDescription	Genres, Songs by genre, null}

MusicPlayerActivity
onMediaItemSelected

item	
mDescription	Jazz & Blues, Jazz & Blues songs, null}

MusicPlayerActivity
onMediaItemSelected

item	
mDescription	Jazz in Paris, Media Right Productions, Jazz & Blues}

```
/** @look doppio/50fe0221-ac36-45e3-8a70-d1a16dd77ae6 */
    @Override
    @DebugLog
    public void onMediaItemSelected(MediaBrowserCompat.MediaItem item) {
        LogHelper.d(TAG, "onMediaItemSelected, mediaId=" + item.getMediaId());
        if (item.isPlayable()) {
            showWarning(item.getMediaId());
        } else if (item.isBrowsable()) {
            navigateToBrowser(item.getMediaId());
        } else {
            LogHelper.w(TAG, "Ignoring MediaItem that is neither browsable nor playable: ",
                    "mediaId=", item.getMediaId());
        }
    }
```

MusicPlayerActivity
onClick

dialog	android.support.v7.app.AlertDialog@e4
which	-1

PlaybackControlsFragment
onClick

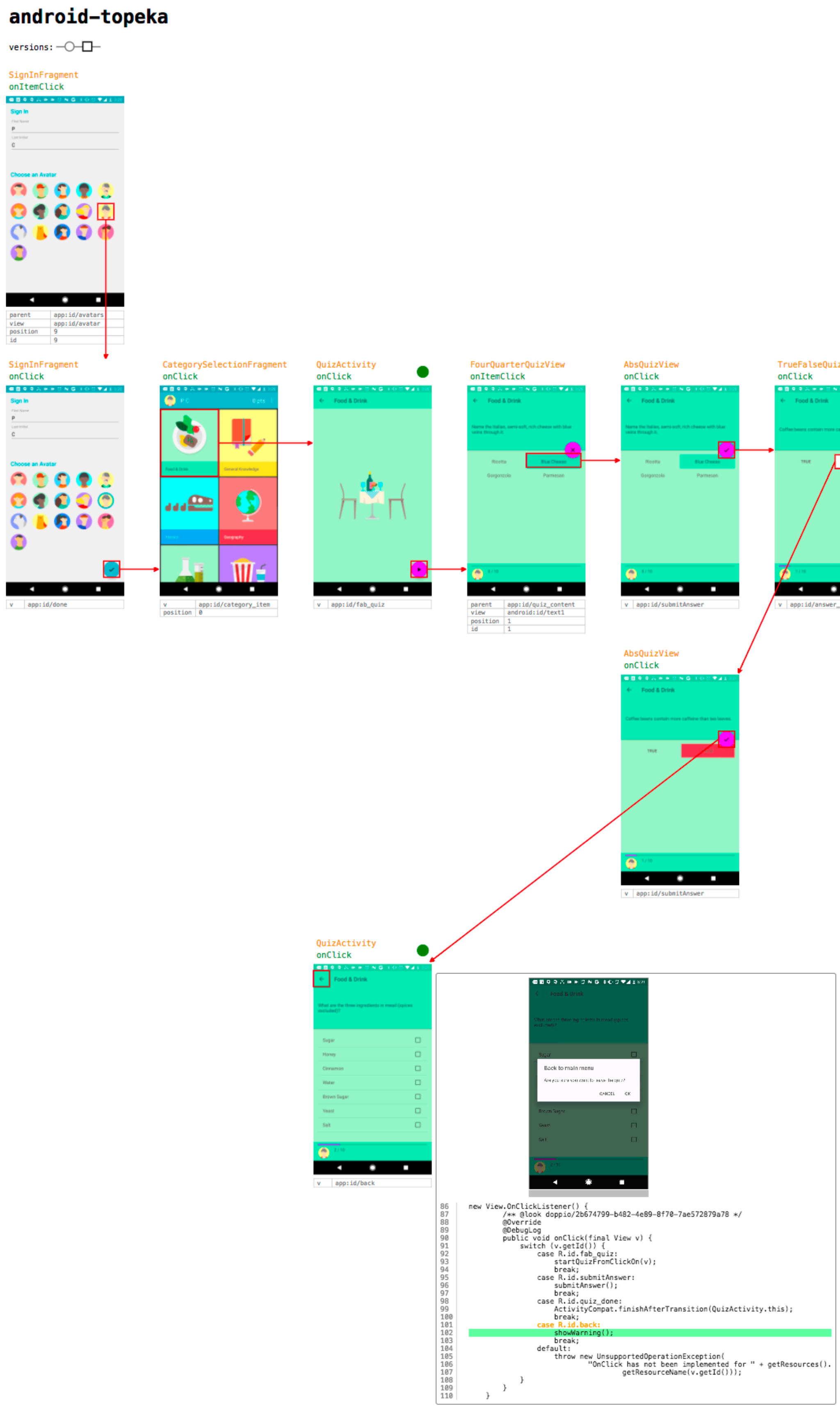
v	app:id/fragment_playback_controls

FullScreenPlayerActivity
onClick

v	app:id/play_pause

ActionBarCastActivity
onBackPressed

Appendix VII. Final screenflow diagram shown in the Viewer for Task 2 after code modification in Study 2b.



```
new View.OnClickListener() {
        /** @look doppio/2b674799-b482-4e89-8f70-7ae572879a78 */
        @Override
        @DebugLog
        public void onClick(final View v) {
            switch (v.getId()) {
                case R.id.fab_quiz:
                    startQuizFromClickOn(v);
                    break;
                case R.id.submitAnswer:
                    submitAnswer();
                    break;
                case R.id.quiz_done:
                    ActivityCompat.finishAfterTransition(QuizActivity.this);
                    break;
                case R.id.back:
                    showWarning();
                    break;
                default:
                    throw new UnsupportedOperationException(
                            "OnClick has not been implemented for " + getResources().
                                    getResourceName(v.getId()));
            }
        }
    }
```