A Qualitative Study on Framework Debugging

Zack Coker*, David Gray Widder*, Claire Le Goues*, Christopher Bogart*, and Joshua Sunshine*

***Abstract*—Features of frameworks, such as inversion of control and the structure of framework applications, require developers to adjust their programming and debugging strategies as compared to sequential programs. However, the benefits and challenges of framework debugging are not fully understood, and gaining this knowledge could provide guidance in debugging strategies and framework tool design. To gain insight into the framework application debugging process, we performed two human studies investigating how developers fix applications that use a framework API incorrectly. These studies focused on the Android `Fragment` class and the ROS framework. We analyzed the results of the studies using a mixed-methods approach, using techniques from qualitative approaches. Our analysis found that participants benefited from the structure of frameworks and the pre-made solutions to common problems in the domain. Participants encountered challenges with understanding framework abstractions, and had particular difficulty with inversion of control and object protocol issues. When compared to prior work on debugging, these results show that framework applications have unique debugging challenges.**

***Index Terms*—Frameworks, Debugging, Qualitative study**

I. Introduction

Frameworks are an important and challenging part of the modern software development process, illustrated by StackOverflow where 7 of the top 24 questions categories were on frameworks.[1]. However, to the best of our knowledge, the prior literature only briefly mentions the challenges that arise during the framework application debugging process. At best, prior work discusses general debugging principles that apply to framework application debugging only at a high level, such as developers using keywords in an error message to find the correct section of code to fix [1]. Some examples of prior literature include a study that categorized the high-level learning barriers of new framework developers [2] and another that investigated how developers search for relevant information while debugging [1]. Researchers have investigated how aspects of the debugging process conform to information foraging theory [3], found that framework application developers are willing to wait a long time for help on Question and Answer forums [4], and found that framework applications contain code patterns [5], [6].

Our conjecture is the features that differentiate framework programs from sequential programs that use libraries (the heavy use of inversion of control, object protocols, and declarative artifacts [7]) present unique debugging challenges and unique debugging benefits. As examples, *inversion of control*, where core framework code controls the data and execution flow of an application [7], could increase the complexity of understanding whether and how state changes in one method call affect another method call, while the standard structure of framework applications could make information easier to locate. A better understanding of how these factors influence the debugging process could lead to improved framework design, along with improved strategies and tools for debugging.

This paper presents an exploratory study that seeks to understand how frameworks specifically both help and hinder developers during the debugging process. To improve our study's external validity, we investigated two frameworks with different use cases: Android, a mobile development framework, and the Robotic Operating System (ROS), a robotics framework. We created debugging tasks by creating violations of framework *directives*, statements in framework documentation that specify prescriptive guidance about the framework's application programming interface (API) and usually contain nontrivial, unexpected information (e.g., "[`setArguments`] can only be called before the `Fragment` is attached to its `Activity` ") [8]. We focused on directives because these statements surface programming challenges specific to interacting with a framework, instead of those related to a particular application. We collected directives for these frameworks and then created debugging tasks based on directive violations (code that contravenes a directive). We then had participants perform these debugging tasks and recorded their debugging process.

We used a mixed-methods approach to guide our study and analysis, borrowing techniques from case studies [9], constructivist grounded theory [10], and qualitative content analysis [11]. We used this approach to draw insights from debugging cases of interest, while making the cases as realistic as possible. We found that certain aspects of frameworks aid developers while debugging, while other aspects of frameworks present challenges (e.g., inversion of control causes participants to misdiagnose method states). Our key contributions are:

- Released summaries from two human studies of debugging directive violations in the Android `Fragment` class and the Robotic Operating System (ROS).
- A categorization of how violations of framework directives are presented to developers
- An enumeration of the benefits and challenges in debugging misuses of framework APIs.

We present the human study methodology in Section II, and the analysis and results of that study in Section III. Section IV outlines threats to validity. Section V discusses related work. Section VI concludes.

* all authors are from Carnegie Mellon University

[1]As of Aug. 14th, 2018

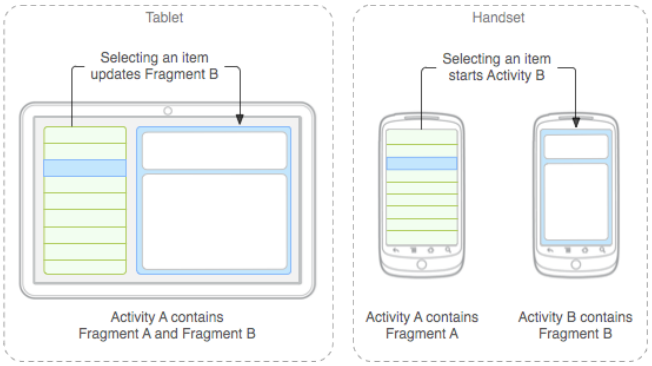

Fig. 1: An example of the `Fragment` class taken from the Android development documentation. This diagram demonstrates how the `Fragment` class is used in an `Activity`.

II. Methodology

We performed an exploratory study into framework API debugging to understand the unique aspects that framework application developers encounter during the debugging process. Our exploratory study consisted of multiple participants performing lab-created debugging tasks. The risk of this approach is that the conclusions may not apply to real framework debugging scenarios. To mitigate this risk, we took steps to make the tasks as realistic as possible, basing the tasks off debugging scenarios encountered by actual developers in practice. We describe the methodology we used to select the frameworks to study Section II-A and create the tasks in Section II-B. The results from an investigation into the possible types of directive violations, used to inform the chosen tasks, are presented in Section II-C. Once we created the tasks, we recruited study participants and conducted human trials, using the procedure described in Section II-D. After recording participants performing the tasks, we coded the recordings using an iterative process. First, we coded the interesting actions of the first few participants in each framework that provided insight on the problem solving process — for example, we recorded if a participant ran the debugger or made a problem related statement but we did not include if a participant accidentally clicked on a wrong tab. Then, we defined coding frames of the interesting actions using qualitative content analysis, a technique that condenses verbal or visual data into important topics [11], and used the coding frames to code the recordings. After we finished coding, we performed theoretical sorting to condense the coded data into the benefits and challenges of framework debugging [10]. A summary of our coding process is described in Section II-E.

A. Frameworks in the Study

Frameworks provide a set of interfaces and classes that reduce the cost to achieve a general goal [12]. Developers create applications to achieve specific goals by extending frameworks, often by extending abstract framework methods. The framework typically calls application code through *inversion of control*, a design in which the core framework code, not the application-specific code, controls the data and execution flow of an application [7]. Frameworks commonly require applications to conform to a specified application structure. Frameworks also commonly impose *object protocols*, or ordering constraints on calls to an object's methods [13], and require certain *declarative artifacts*, non-source code files that contain configuration information [7].

We constructed debugging tasks for two frameworks: Android (v. 5.0 Lollipop–API 21, specifically the `Fragment` class), and the Robotic Operating System (ROS) (Kinetic Kame).

Google's *Android* [14] provides a Java framework for developing mobile applications. Android is a widely-used, mature framework and has been released for over seven years. In the Android framework, the `Fragment` class represents a reusable component of an Android application's user interface. A picture of an Android `Fragment` in an example Android application is shown in Figure 1,[2] to illustrate its usage. We began with the Android framework for three reasons: (1) it is widely-used and well-established, (2) it makes heavy use of inversion of control and object protocols, key features that differentiate frameworks from libraries, and (3) multiple developers express difficulties with the framework, as demonstrated by the many questions on StackOverflow about it.

After selecting the Android framework, we decided to perform an in-depth analysis into the directives of one class, to collect a diverse set of directive violation consequences. We queried StackOverflow for questions by class, finding that the `Activity` class had the highest number of posts and the `Fragment` class had the second highest number of posts. After investigating the questions for both, we focused on the `Fragment` class because the *Activity* keyword often turned up questions that were not related to the class. The `Fragment` class also requires developers to correctly implement the `Fragment` lifecycle inside of the `Activity` lifecycle, a fairly complicated workflow, which we speculated might lead to more or more interesting debugging challenges.

To support the generalizability of our claims, we selected ROS as a second framework for study. The *Robot Operating System* [15] (ROS) is a framework for creating robotics applications, with a focus on the communication between various robotic components. ROS applications are built as a collection of nodes that communicate in an event driven model. ROS is also a mature framework and has been released for over eight years. ROS is written in both C++ and Python. Our criteria for the second framework was that it should focus on a substantially different domain than Android, and have a different framework architecture. ROS satisfies these criteria, because: (1) ROS is designed for robotic applications instead of mobile applications, (2) ROS uses an event based architecture instead of the tiered architecture of Android, (3) ROS is written in C++ instead of Java, and (4) ROS has a smaller user base, but still sufficient users that we could find experienced participants for the study.

[2]Taken from https://developer.android.com/guide/components/fragments

B. Task Creation Methodology

We used violations of framework directives to create the debugging tasks. *Framework directives* are possibly surprising statements in a documentation source about how to use the framework (e.g., "`setHasOptionsMenu(true)` must be called to execute an overridden `onCreateOptionsMenu` method") [16]. Focusing on directive violations was key to our study because violations thereof capture mistakes specific to framework programming rather than bugs related to application logic. This moreover improves the chances that our results generalize to other framework API misuse errors. Framework directives are also likely to illuminate debugging situations where participants have difficulty with a framework, due to their surprising nature.

The process of extracting directives for both frameworks consisted of one author extracting directives from the documentation and another author double-checking the extracted directives, similar to the coding process in prior work [17]. We were lenient on the definition of "possibly surprising" and included any rule that was not already checked by the type system or did not focus on the possible null value for a parameter. However, we restricted the directives in the study to directives that were testable. Thus, in our study, a directive violation is a section of code or application that does not conform to the testable specification statement.

To inform our task selection, we investigated the consequences of violating Android `Fragment` directives and how those violations were presented to developers. We collected 45 Android `Fragment` directives from three official documentation sources: (1) 11 from the `Fragment` page in the developer's guide,[3] (2) 19 from the `Fragment` API page,[4] and (3) 15 from the `Fragment` class's source code. We created violation scenarios for each directive through a manual process of violating the directive and then confirming the directive violation, either through the scenario's output or through print statements. We then categorized the directives by the directive violation consequence, or the effect on the application that the developer sees when that directive is violated. The directive violation consequence categories are discussed in Section II-C.

To select tasks, we searched for StackOverflow questions that cover Android `Fragment` directives from a wide range of violation consequence categories. We used keywords taken from the directives or error messages produced by an author-written violation of the directive to find the StackOverflow questions. We found seven unique questions that covered the directives and provided enough information to reproduce the problem. These seven questions were used to create seven Android tasks. The seven tasks were created by taking an Android Lollipop sample application[5] that demonstrated the various notifications available in Android Lollipop and changing the application to encompass the scenarios mentioned in the StackOverflow questions.

For the ROS framework, we extracted 28 directives from two sources: (1) 9 from the official ROS C++ documentation,[6] and (2) 19 from ROS C++ source code.[7] Due to the relatively low number of online questions about ROS, we were unable to collect ROS directive scenarios from StackOverflow. Instead, we choose three directives that represented materially different cases, and manually created tasks for each. We created the first and third task by modifying the TurtleSim scenario,[8] a two node configuration where a virtual turtle in one window moves and publishes the movements so the virtual turtle in the other window mimics the movements. The second task involved a simple, custom-built directory reading application.

In the rest of the paper, the Android tasks and participants will be prefixed with a "TA" and "PA" respectively. The ROS tasks and participants will be prefixed with a "TR" and "PR" respectively. Table I lists the number of participants per task and briefly explains the Android and ROS tasks. A link to the Android tasks and steps to recreate the ROS tasks can be found in the readme of https://github.com/cbogart/ViolationOfDirectives.

C. Directive Categorization By Consequence

To inform the trials in our human study, we investigated the ways that frameworks present errors to developers. This investigation provided a greater understanding of the debugging situations that developers encounter when debugging framework applications, and ensured a diversity of tasks for our human study. For the investigation, we manually violated Android `Fragment` directives, and then grouped the violation consequences into categories. We then collected three ROS directive violations and found that they fit into the previously created categories. Due to the limited investigation, we do not make claims that the categories will generalize to all frameworks. However, the results are useful as an initial investigation into the ways that frameworks present directive violations to developers.

The consequences of violating 41 Android `Fragment` directives are shown in Table II and explained below. We found that violating certain directives can produce multiple consequences (e.g., a violation can produce a tool warning and crash with reference to the directive), but each consequence is mutually exclusive (the same consequence could not be categorized in multiple categories - an application crash cannot be both classified as crash with reference to the directive and crash without reference to the directive). We could violate one directive in two different ways to produce three possible consequences. We were also able to violate two directives in two ways, each with different consequences.

Compiler Error. When these directives were violated, the framework threw a compiler error, preventing the application from compiling. This consequence occurred when invalid semantics produced a directive violation. One example is when

[3] developer.android.com/guide/components/fragments.html
[4] developer.android.com/reference/android/app/Fragment.html
[5] github.com/googlesamples/android-LNotifications
[6] wiki.ros.org
[7] docs.ros.org/api
[8] http://wiki.ros.org/ROS/Tutorials/UsingRxconsoleRoslaunch

Task	Count	Goal	Violated Directive	Result of Directive Violation
TA1	5	The participant must connect user inputs to the output message when input components initially share the same ID.	Application components must have a unique ID to be referenced individually.	Any attempt to access one of the components with the same ID returned the last component added.
TA2	6	The participant must display the application start time on a tab without a warning.	The application should not pass time data through the constructor.	AndroidStudio displayed a warning and recommended a fix.
TA3	4	The participant must make the framework check for an updated `OptionsMenu`.	The framework only checks for an `OptionsMenu` if the application calls `setHasOptionsMenu(true)`.	The `OptionsMenu` does not appear, although an `OptionsMenu` is defined.
TA4	5	The participant must display the application's `Activity` (the entry point for an Android application) title in a pop-up message on a specific tab.	The `Fragment` could only access the `Activity` if the `Activity` was attached to the `Fragment`, and the `Activity` was not attached.	The application crashed with a notification that the `Activity` was not set.
TA5	4	The participant must fix a problem that occurs when the application tries to change the color of a button on a tab when the tab had been previously accessed.	A tab's arguments can only be set *before* the tab is accessed.	The application crashed, stating that arguments can only be set before the tab has started.
TA6	3	The participant must change a specified `ContextMenu` to an `OptionsMenu`.	Items should be added to the `OptionsMenu` in the `onCreateOptionsMenu` method.	The `OptionsMenu` would not appear.
TA7	5	The participant must fix an incorrect `inflate` method call.	In the application's current state, the last parameter of the `inflate` call must be `false`.	The application crashed with a stack trace that pointed towards core framework code.
TR1	8	The participant must fix an incorrect `spinOnce()` call.	`spinOnce()` cannot be used when the framework should perform the callback more than once.	A node in the application would quit unexpectedly without an error message.
TR2	8	The participant must fix an application node's parameter access.	Local namespaces are not checked if a global namespace is used in a parameter search.	The parameter search returned that the parameter does not exist.
TR3	6	The participant must fix an obsolete message type.	An incorrect message type was used for this version of ROS.	The application crashed with an incorrect type declaration error.

TABLE I: Tasks in our human study. TA tasks indicate Android tasks; TR, ROS tasks. "Count" shows number of participants per task. "Goal" indicates task success conditions. "Violated Directive" is a simplified explanation of the violated directive motivating the task. "Result of Directive Violation" explains how the application presented errors to participants.

Directive Violation Consequence	Count
Compiler Error	3
Crash With Reference To Directive	19
Crash Without Reference To Directive	2
Missing Feature	9
No Obvious Effect	5
Tool Warning	3
Wrong Value Returned	2

TABLE II: Categorized consequences from violating 41 directives. A directive violation may have multiple consequences, but each consequence is mutually exclusive.

the documentation specified that a method could not be overridden. The compiler prevented a developer from overriding this method because the method was declared with the final modifier in the parent class.

Crash With Reference To Directive. When these directives were violated, the application crashed with an exception that notified the user of the directive violation either directly or indirectly. One example of this category is, *`getActivity()` should not be called when the `Fragment` is not attached to the `Activity`.* If this directive was violated, the application crashed with a null return from `getActivity()`. This category contains a high number of directives because all the directives found in the `Fragment` class's code were of this type.

Crash Without Reference To Directive. When these directives were violated, the application crashed with an exception that did not notify the developer that a directive was violated, usually with an error pointing to where the application crashed instead of the location where the application needed to be fixed. Violations in this category occur when a more general exception message is thrown, or violating the directive puts the application into an invalid state and the invalid state is caught in a later line. One example in this category is *when the result of the `inflate` method is used as the return result for `onCreateView`, the last parameter to the `inflate` method call must be `false`.* If this directive was violated, the application would crash with a stack trace that pointed to internal framework code and not the `inflate` line.

Missing Feature. When these directives were violated, the

framework application did not produce the intended effect of the relevant section of code (i.e., the code ran without errors but the section of code with the directive violation did not produce the intended feature). The effect did not occur either because violating the directive caused the control flow to change or the semantics were changed. For example, one directive states that *an application will only execute the* `Fragment`*'s* `onCreateOptionsMenu` *method if the* `Fragment` *calls* `hasOptionsMenu(true)` *in the* `onCreate` *method*. If the `hasOptionsMenu(true)` call is removed, the `OptionsMenu` will not appear, even if the `Fragment` overrides `onCreateOptionsMenu`.

No Obvious Effect When these directives were violated, the framework correctly performed the intended action of the associated code segment without crashing the application. One example of this category is a directive that states that *if a* `Fragment` *does not have a user interface (UI), then the* `Fragment` *should be accessed by* `findFragmentByTag()`, but the `Fragment` without a UI could be accessed by `findFragmentById()` without noticeable consequences.

Tool Warning. While the application still compiled with these directive violations, if these directives are violated, the recommended tools for developing applications in the framework (in the case of Android, AndroidStudio, the recommended Android Integrated Development Environment) warn developers about the code, and sometimes recommend a possible fix. For example, the directive, *classes that subclass the* `Fragment` *class must have a public no-argument constructor* produces a warning when violated. An application will compile if the class subclassing `Fragment` lacks an empty constructor, but AndroidStudio displays a warning in the class's source file.

Wrong Value Returned. When these directives were violated, the application did not crash, but a reference to a part of the application was lost or used incorrectly. Any attempt to use the lost or incorrect reference returned a wrong value. For example, *when a developer dynamically added a UI element, the developer must assign a unique tag to the added UI element.* If the added UI element does not use a unique tag, the new tag overrides the matching tag of a previous UI element. The previous UI element is now unreachable through framework supported methods.

D. Human Trials Methodology

We based our human trial methodology on other previous developer studies, such as study by Piorkowski et al. [18]. After we obtained IRB approval, our human trial process started with a pre-survey to document participants' framework experience. We provided participants a Surface Pro 3 tablet containing the tasks, and we instructed them to perform think-aloud debugging, vocalizing what they thought as they went through the debugging process [19].

We assigned a task to each participant, and asked them to fix the bug. We did not inform participants of the directive violation in the task because we were interested in also studying the fault localization process. If participants finished a task and could stay for another 20 minutes, we asked them to attempt another task. We did this to gain more insight on the question if certain tasks were hard or if certain participants were much more efficient at all the tasks. All but one participant were willing to spend more than 20 minutes to attempt the task once they started. We initially assigned tasks randomly, but later selected tasks that the fewest participants had attempted, to produce relatively even task coverage. Android participants spent about two to three hours for each session. Due to removing an unhelpful pre-trial application familiarization period, our ROS sessions lasted around one hour. Participants were permitted to quit at any time (we never stopped them in their work). We allowed participants to search online for anything, including the inspiration for the tasks, but we did not allow them to post questions. While searching online, no participant found the inspiration for any of the study's tasks. During think-aloud debugging, if participants stopped explaining their thought processes, we prompted them through questions like *"What are you thinking now?"* and *"What is your reasoning for that?"* Occasionally, we asked *"Why not?"* if participants commented they would normally do an action, but they were choosing not to. In addition to asking them to vocalize aloud their thoughts and strategies during the tasks, we asked participants about their approach in greater detail at the end of each participant's session.

For the Android study, we collected a convenience sample of 15 participants. Eleven of the participants had over 2 years of industrial Java or Android experience, and 14 of the participants had more than a year of industrial Java or Android experience. Two participants were professional developers, and 13 were graduate students (12 with professional background). For the ROS study, we collected a convenience sample of 12 participants. Nine of the 12 participants preferred the C++ version of ROS over the Python version. Two of the participants had more than 2 years of ROS experience and 5 of the participants had over a year of experience. Three of the participants were research staff, 8 of the participants were graduate students, and 1 was an undergraduate student. None of the participants did both the Android and ROS study.

We made several procedure changes between the two case studies. For Android, we gave participants time to learn the application before attempting the tasks, while we did not provide a learning period for the ROS tasks. We made this change because we found that participants commonly spent the Android learning period exploring sections of the application that were not relevant to the tasks. In the Android study, we required participants use the recommended Android Integrated Development Environment (IDE), AndroidStudio, because it provides warnings for directive violations. We did not require participants to use any particular IDE for the ROS tasks, because ROS does not have a recommended IDE.

E. Coding Process

To analyze the results of the human studies, we coded the actions and statements of participants from our video (participants' screens during the trials) and audio (all audio

from the trials) recordings. We followed an iterative, open-coding practice based on techniques from Qualitative Content Analysis [11] and Constructivist Grounded Theory [10]. We made adaptations to the techniques, which are described for interview-focused studies, to render them suitable for our human trial method of data collection.

The iterative aspect of the coding process involved analyzing recordings of the first few participants for important concepts before conducting debugging trials with the remaining participants. After analyzing the recordings from these participants, we then conducted more debugging trials. We alternated between conducting and analyzing trials, and adjusted our coding frame based on the new trial results. As recommended by Contructivist Grounded Theory, we tried to reduce the impact of prior work on our coding process (to avoid analyzing the data with a pre-determined code and thus possibly bias the results) [10]. We thus limited our initial reading of prior work to a few similar studies, and investigated the rest of the related work after finishing the analysis.

Our coding process consisted of two different authors coding different aspects of the study. One author conducted and coded the Android trials while another author conducted and coded the ROS trials. We allowed the coding categories for both the Android and ROS study to emerge independently, although both were checked and guided by an expert in the area of frameworks (an author), which led to variations in the categories and granularity of the different coding categories.[9]

The Android coding categories consisted of six categories that focused on the *mental aspects of the debugging process* ('deductions,' 'questions asked,' with a subcategory of 'hypothesis creation,' and 'hypothesis rejection') and the *actions* that participants took ('coding/testing actions,' and 'activities taken to learn the framework or code base'). Finally, a miscellaneous category included behaviors such as reading the scenario specific prompt, or a participant expressing an opinion (such as annoyance with the lack of documentation on a topic). The ROS coding categories consisted of four major categories: 'goals,' 'activities,' 'learning,' and 'miscellaneous.' Each category contained multiple sub-categories, for example, the subcategories under 'goals' were 'fix an identified problem,' 'answer a question,' 'test a hypothesis,' 'find a file,' or 'other goal' (not included in the previous four). The coding categories of both studies were refined over time. For example, the Android coding categories initially contained a 'goal' category, but this was eventually merged with the 'questions' category as participants often expressed goals in question form.

One author then condensed the coding data from both studies into the final categories of framework benefits and challenges using theoretical sorting. There are multiple acceptable approaches for theoretically sorting data in constructivist grounded theory [10]. We choose to categorize the results into the benefits and challenges of framework debugging, as this frame provided the most insight.

[9]While our IRB approval prevents releasing the recorded trials, we have released our coding categories along with our identity-scrubbed and coded trial summaries to promote future researchers reproducing our results at https://github.com/squaresLab/frameworkStudyTranscripts

III. Framework Debugging Benefits and Challenges

We first present the challenges that developers faced while debugging the tasks: those relating to dynamic behavior (Section III-A), static structure (Section III-B), and historical changes to the framework (Section III-C). Next, we present the benefits of framework debugging: dynamic benefits (Section III-D), static benefits (Section III-E), and historical benefits (Section III-F). Finally, we discuss the difficulty of debugging violations in relation to their consequence category as an inspiration for future work (Section III-G).

A. Dynamic Challenges

Throughout the study, participants struggled to determine the order in which a framework executes application code, which increased the difficulty of the debugging process. Participants seem to prefer a cause and effect ordering. However, framework code does not typically follow a sequential ordering, instead executing application code only when needed. This requires code to be structured as non-sequential event handlers. This can create uncertainty about which parts of project-specific code are called and when, and which project method will execute "next." Object protocols exacerbate this issue by requiring participants to understand which states various objects can be in when the framework calls their code.

Inversion of Control. In framework programs, application-specific method execution order is not always transparent to the application developer. This sometimes led participants to misunderstand an application's control flow, which increased the difficulty of locating the error and fix locations. For example, in the ROS study, participants (PR18, PR20) assumed the framework did not call a section of code, when instead a problem in that code segment caused the application to terminate earlier than expected. In Android, two participants (PA10, PA11) tried to use the debugger to understand control flow, but struggled to do so. Both participants stepped past an application method and were unable to figure out how to step back into non-framework code. This led participant PA10 to reach incorrect conclusions about which code executed in task TA7. In ROS, while trying to understand how two nodes communicated, PR22 did not realize that a third node linked two other nodes, because the nodes relied on the framework to handle communication. Participant PR22 read four files before understanding how the framework routed the nodes' communication. Another participant (PR23) made incorrect control flow deductions due to the way ROS redirects and filters statements printed to standard output.

Inversion of control also made localizing errors difficult. In Android, one participant (PA5) searched for an error message thrown by the application, but could not find it in the project. The search failed because the error message was generated from core framework code, not project code.

Some problems stemmed from participants' uncertainty about the hidden ordering of critical framework activity between events. When participants (PA4, PA12) saw the `getActivity` call returned `NULL`, they questioned whether the framework had incorrectly constructed its own reference to the parent `Activity`. In fact, `getActivity` was called in an event that occurred before the framework had attached the `Activity` to the `Fragment`.

Takeaway: The inversion of control in frameworks increases the difficulty of understanding the application's control flow, which increases the difficulty of debugging the application.

Object Protocols. Object protocols are object states that dictate how an object can be used. An example of object protocols in Android are lifecycles: state transitions between starting, active, and stopping for components. Participants experienced challenges with object protocols (e.g., accessing values before they were set). Object protocols are explained and diagrammed explicitly in the documentation, but implemented indirectly in the framework code, and invisible to non-framework code. This likely increases the difficulty of understanding the relevant object protocols.

Object protocol issues in tasks TA4 and TA5 significantly contributed to the amount of time those tasks took (see Section III-G). Most participants assumed the application had performed an invalid action, rather than that an action that was invalid only in a given state. Object protocol misunderstandings also led participants to incorrectly conclude that certain values were available for application use. Three participants (PA4, PA6, PA11) mistakenly wrote code to access variables storing user selections before the user could have selected them. Participants were then confused when the accessed values did not match those they selected in testing. Participants (PA6, PA10) were confused about the circumstances in which they needed to commit and finalize a `Fragment` transaction (as opposed to the cases in which transactions were automatically committed). PA1 expressed exasperation on the difficulty of keeping track of when methods were called "You look at the Fragments documentation and it's like lifecycles inside of lifecycles inside of other lifecycles. It's annoying. It's like, how do I keep track of all this?" In the ROS study, participant PR26 mentioned uncertainty about how to modify a method because of the states the application could be in when that method was called.

Takeaway: The way frameworks hide object protocols causes problems when diagnosing the problem location and when producing the correct fix.

B. Static Challenges

The static structure of frameworks, such as declarative artifacts and the mapping between source files and executable components, presented multiple challenges to participants. Participants commonly struggled to understand the separation between static structure and dynamic changes, determine the effects of the application's static configuration, and use that knowledge to solve problems. This led to uncertainty about whether errors should be addressed by modifying static files, or via a dynamic solution. Participants also often struggled with determining the correct framework terminology for their situation, which increased the difficulty of finding related documentation and online solutions.

In ROS, participants had difficulty understanding how source files map to executable components, partially because ROS executables consist of various components (Nodes, Services, and Topics) that do not map directly to source. ROS also does not provide an easy or well-known way to find source code corresponding to a given component. Participant PR16 struggled to understand how the publisher and subscriber methods in the C++ files integrated with the data redirections in the launch file. Other participants (PR17, PR18, PR20, PR22, PR25, PR27) similarly struggled to understand how the application remapped data between components. In one case, Participant PR17 diagnosed the problem but struggled to find the appropriate source file, due to both the organization of the ROS application files and their confusion about how to use the ROS filesystem commandline tools: "I am looking for source code for [this node]... Unfortunately ROS is trying to isolate me from the file system, which I dislike, because it cannot isolate me fully." Some 16 minutes into the task, Participant PR17 exclaimed "This is ridiculous, I can't even find the code that I am supposed to be debugging!" They eventually used `grep` to find a node, more than half an hour into the task.

Multiple participants were confused about which framework concept applied to the current task. For example, both TA3 and TA6 asked participants to add an `OptionsMenu` to the application. Participants were given an application were the `OptionsMenu` did not appear and a picture that showed the application's `OptionsMenu`. Participants were asked to fix the problem in the application so that the `OptionsMenu` appeared. Android's `OptionsMenu` appears on the `ActionBar` but is not managed by the `ActionBar` class directly. At the beginning of the task PA9 commented "I believe that is probably related to the ActionBar, because I think that is where the user defined menus can go." PA9 ended up performing three different searches related to icons in the `ActionBar` which all initially looked promising to the participant but were ultimately unsuccessful. The participant's fourth search, *linearlayout action bar*, found a site that directed the participant to add the options through the `OptionsMenu` and lead to a successful completion of the task. PA1, PA2, PA3, PA9 all demonstrated difficulty determining the correct framework terms and concepts related to the issue.

Participants in the study were often unsure if changes to the application were performed dynamically with method calls or statically with adjustments to options in a *declarative artifact*, such as the XML layout specifications in Android. In the same tasks (TA3 and TA6), many participants (PA9, PA13, PA14, PA15) were initially confused if the `OptionsMenu` was added through a declarative artifact or through a method call. These participants looked through the Android layout editor for an `OptionsMenu` or tried to add an `OptionsMenu` to a XML file, before realizing that it must be added dynamically. Another participant (PA9) investigated the `strings.xml` file after an online answer suggested that the problem may lie in

an undefined icon title. Participant PA10 remembered that a specific theme could cause errors and checked if the theme caused the error.

Takeaway: Unclear mappings between declarative artifacts and the application's source code provides a source of debugging difficulty.

C. Historical Challenges

The fact that frameworks change over time can increase the difficulty of debugging framework errors: both participants' knowledge and online help or solutions may be out of date.

Legacy Challenges. Previous versions of both Android and ROS created issues for several participants. In Android, one participant (PA9) questioned whether a feature should be implemented in a backwards-compatible way, later discovering that the application was not configured to work with backwards-compatible components. A few participants (PA8, PA9, PA14) avoided online answers older than two years because they assumed they would no longer apply. Other participants (PA1, PA15) mentioned they were familiar with Android a couple years ago, but there had been many changes to the framework since they were proficient with it.

Some ROS participants incorrectly diagnosed the obsolete message type in task TR2 as correct because they had used it previously. Participant PR21 recognized that the message type caused an issue, searched the message type online, and found its official documentation, not realizing that the documentation was for an older ROS version. This was a problem because the documentation indicated that the file was using the message type correctly. The participant investigated four other possible error sources before realizing that a different message type was needed.

Past Experience. While past experience was often helpful, one participant in the Android study (PA10) misdiagnosed an error message due to previous experience. This caused the participant to conclude to "not trust your experience."

Takeaway: Out-of-date knowledge about a framework and similar error messages that cause developers to incorrectly diagnose the problem increase the difficulty of debugging framework applications.

D. Dynamic Benefits

Throughout the study, participants commonly used the framework to perform actions that would have been much more difficult to recreate without the help of the framework. When faced with a task, almost all participants tried to implement the correct, prescribed framework method for performing the required action (although PA1, PA3, PA5, PA8, and PA11 implemented custom solutions for certain tasks, such as implementing a custom message passing solution in TA1). For example, PA6 in TA1 correctly used the `FindFragmentById` method to access user input, instead of writing code to pipe the data through the application. Overall, developers can take advantage of the benefits that framework methods provide.

Takeaway: Frameworks provide methods to address common task in a framework application. These methods can decrease the difficulty of implementing a bug fix.

E. Static Benefits

Study participants found the static organization of the framework helpful when trying to gain an overview of the application, which helped them find files of interest more easily than through unstructured search. In ROS, participants used the launch files as a way to start exploring the application. For example, participant PR27 looked through the ROS launch files to understand which nodes were involved in the application. Participant PR26 mentioned that they liked to use launch files to get an overview of the application. Multiple participants (PR17, PR18, PR19, PR22, PR26, PR27) used the launch files as a table of contents, using them to determine application components, how they interact, and locate source files of components.

Similarly, in Android, participants (PA1, PA2, PA3, PA6, PA8, PA9, PA10, PA11, PA13, PA14, PA15) used the structure of Android application to quickly find resource files and test case files. For example, PA8 was able to quickly look up the correct options menu layout file when writing the required options menu code. None of our tasks required participants to resolve bugs in declarative artifacts, so it's possible that this type of problem would be especially easy to debug.

Takeaway: The common structure of frameworks applications reduce the time to find necessary information when debugging.

F. Historical Benefits

Participants often found that past experience was helpful; some were able to correctly diagnose a ROS error simply by looking at the failing section of code and relating it to code or problems they had seen before. Multiple ROS participants (PR17, PR21, PR26, PR27, PR28) were able to diagnose an error and suggest a working alternative based on past experience. While working on task TR2, participant PR28 noticed the error in the code and said, "I think the fact that there's a beginning slash means that instead of looking under this node's namespace it's gonna look under the global namespace [where] this parameter doesn't exist." The participant was correct. Detailed knowledge of a framework, built up by through experience, can help mitigate barriers frameworks impose. Other participants (PR18, PR22, PR26, PR27) stated that past experience shaped their general ROS debugging strategy. One participant remembered to set framework environment variables, attributing past environment problems. Another participant (PR26) always used `grep` to find calls to a function modified over the course of a debugging session, to guard against unforeseen side effects, a problem they had faced in the past.

Takeaway: The common elements of applications created in a framework allows developers to build debugging experience in the framework.

Violation Consequence	Time (Mean)	Sessions Completed	Sessions Attempted	Success Rate (%)	Tasks
1. Android: Wrong Value Returned	51 min	4	5	80	TA1
2. Android: Crash With Reference To Directive	47 min	3	9	33	TA4, TA5
3. Android: Missing Feature	28 min	4	8	50	TA3, TA6
4. Android: Tool Warning	23 min	6	6	100	TA2
5. Android: Crash Without Reference To Directive	19 min	4	5	80	TA7
6. ROS: Missing Feature	49 min	5	8	63	TR1
7. ROS: Wrong Value Returned	36 min	5	8	63	TR2
8. ROS: Compiler Error	25 min	6	6	100	TR3

TABLE III: Mean time on task and task completion rate, by consequence. Time on task includes failed attempts.

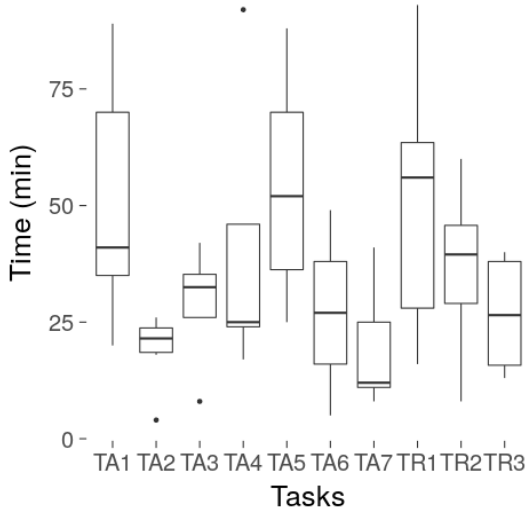

Fig. 2: Time spent per task. Timing includes failed attempts.

G. Difficulty By Consequence

We analyzed participant results from both the Android and ROS tasks using the consequence categories collected from the Android investigation. We validated these categories by creating ROS trials and found that all of the ROS directive violation consequences fit into previously created categories. While we did not perform enough trials of each task for statistical significance, the results from the trials provide insight on the difficulties between the tasks, which is useful for directing research towards the tough framework problems. We present the time participants spent on each task and the participant success rate on each task in Table III. The time participants spent on each violation category is presented as a number not to represent a precisely measured time for the task, but to present a better sense of the data than descriptive terms (i.e., much longer, longer, about the same, shorter, much shorter). Figure 2 shows a box-and-whisker plot of the participant's time spent on the tasks in the study. There was a substantial difference in the mean time to complete tasks (ranging from 19 to 51 minutes) and the success rate on tasks (ranging from 33% to 100%) of difference consequences.

The consequence of violating a directive appears to influence how long it takes to debug the error as well as how likely a developer is to succeed in doing so over a short debugging session. A surprising result compared to prior research on debugging is that participants had significant difficulty with finding a fix once they were aware of the problem location. As an example of prior research in this area, Vessey [20] found that fault localization often took longer than finding the fix. However, developers who worked on the tasks in *Android: Crash With Reference To Directive* category were notified of the fault location immediately, yet this category had one of the longest mean times. Even for other tasks that participants were immediately notified of the fault location, participants did not immediately determine the correct fix, often taking 20 or more minutes (TA2, TA4, TR3). In these cases, participants knew that certain directives were violated but they did not know how to fix the error. Participants found this frustrating, with one participant stating *"Why don't they [the documentation] tell me the right thing to use? They tell me it is going to cause a problem but they don't tell me what the alternative is."*

Takeaway: We observe that it appears important not only to notify developers of directive violations but also to help fix directive violations explicitly.

IV. Limitations

External Limitations. We attempt to mitigate the risk that our results will fail to generalize to other frameworks or languages by investigating two different frameworks and a wide range of framework debugging problems. We have probably not discovered all of the benefits and challenges of framework debugging (or Android and ROS debugging), but we have reduced this risk by first investigating the space of error presentations that developers face (the consequence categories) and then investigating those categories in-depth with the human trials. Our categorization of framework directives by violation consequence may not generalize (e.g., other frameworks may not have recommended development tools), and it may be incomplete; in particular, we did not consider potential non-functional violation effects, such as degraded performance. The lack of completeness could motivate future work.

Our constructed tasks may not represent solving real-world debugging issues. This concern was reduced by basing the Android tasks on StackOverflow questions. Additionally, participants were new to the code in each task, possibly leading to unrealistic code familiarity problems. We sought to reduce this threat by providing Android participants with a learning period, but we note that, for example, one participant

mentioned that it would be preferable to spend a day reading documentation before tackling the tasks. As such, time limitations may have influenced our results. Finally, the participants in the study may not represent the population of framework users, and instead might be biased by the large number of student participants. We attempted to address this limitation by recruiting participants with framework experience: 14 of the Android study participants had over a year of industrial Android or Java experience and 7 of the ROS participants had over a year of ROS experience. It is likely that our results represent the challenges of developers that are moderately skilled in the framework, and may not apply to multi-year experts. The sample sizes of both number of trials in each category and number of tasks in each category are too low of a sample size to draw conclusions with statistical significance. While important in certain cases, our goal was to perform a qualitative-exploratory study, where statistical significance was not a primary concern (goal was to determine what problems developers have instead of how often these problems occur). Achieving statistical significance with number of tasks or participants is left to future work.

Internal Limitations. Participants could freely decide, in a low-risk situation, when to quit a task. Participants were also asked to think-aloud, and prompted to do so by the researcher. These prompts may have altered the approach a participant would have taken absent the prompt. The think-aloud component may affect how long participants took to solve the tasks. We believe that this affected tasks roughly equally, such that tasks which took significantly longer than the others are likely to have taken longer in a non-think-aloud context. While the size of the different problems were restricted to directives, the amount of code required to fix the directive was not consistent across all tasks. However, all tasks could be fixed with ten additional lines or less. Finally, some participants mentioned they would have been more comfortable if the researcher were not watching, and if they were able to use their preferred IDE, OS, or laptop. These irritants may have caused participants to perform differently than they would have in their preferred environment.

V. Related Work

Frameworks and APIs. Ko et al. [2] investigated the challenges that new end-user programmers face (using a framework application as their study subject). They identified learning barriers which summarize broad challenges in this domain, design barriers and use barriers. We expand on this prior work by focusing on and expanding the subset of challenges that apply to framework debugging specifically, and do not restrict our attention to end-user or novice programmers. Other prior work has created formal specifications for framework plugins [21], [22], statically analyzed declarative artifacts [23], looked through StackOverflow to find framework problems [24], and patterns that appear in framework development [5], [6]. Another study that investigated StackOverflow questions on frameworks found that developers had difficulty determining the correct framework term (as we found in our study)[25]. To the best of our knowledge, none of this prior work specifically addresses the process of debugging of framework applications.

Multiple studies have found that API design decisions can significantly impact programmer effectiveness [26], [27], [28], [29]. Our study investigates the challenges developers encounter when using framework APIs. Our results, while not focusing on design decisions directly, can help guide framework designers away from the usability problems we uncovered. Others have found that API changes lead to application problems, when developers do not keep applications up-to-date with API changes [30], [31], and investigated patterns in API updates [32]. This prior work focuses on the challenges of updating applications to conform to API changes, instead of the problems caused by out-of-date knowledge of how to use the API.

Information Needs. With respect to eliciting information while programming and debugging, developers have a wide range of documentation preferences [33]. They search the web in multiple styles [34] and use keywords in bug reports as a guide when searching through source code for maintenance tasks [35]. Developers use more sources of documentation during maintenance tasks [36] and want rationale for changes during code review [37]. Our study expands the community's knowledge of information foraging in framework application, by discussing the challenges developers face when the required information is more fragmented (as in the user written application and the internal framework code).

Ko et al. [1] investigated how developers collect information during software maintenance, focusing in particular on applications built against libraries (rather than in a framework context). As in our study, developers debugging library-based applications often searched for sections of code that seemed relevant to the failure, either based on keywords or nearby functionality. However, because method call ordering is not easily viewable inside a framework application, participants in our study would sometimes fail with this strategy, missing relevant code. For example, in TA4, participants often spent a long time investigating the `OtherMetadataFragment` (usually over 20 minutes) before moving to investigate the rest of the application, likely because they could not easily understand how the rest of the code was connected to the error location. Our findings imply frameworks can add extra complexity to the debugging process as compared to debugging standard library-based applications.

Directives. Prior work has focused on how to classify directives [16], [8], [38], found directive knowledge is helpful during development [16], [8], investigated the prevalence of types of directives [39], fixed mistaken directive documentation using source code [40], and found that directives can be used to answer a sample (16/20) of StackOverflow questions [41]. We expand on this prior knowledge in our focus on the challenges involved in debugging directive violations.

Debugging and debugging theories. Previous studies have investigated what developers want to know when debugging

(including dataflow and references to the problematic section of code) [42], [43], [44]. Others have investigated maintenance tasks in machine learning [45] and cloud deployment software [46]. Additional studies have focused on how design decisions are handled in the repair process [47], how developers use scent finding to locate the fault [3], and how tasking developers with fixing or learning about a bug lead to different navigation behaviors [18]. Our study builds upon these results by qualitatively investigating the unique debugging challenges in a framework application context.

Recent research has demonstrated the importance of the types of bugs investigated in our study by, e.g., examining framework-specific errors in Android applications [48] or proposing to automatically repair crashing Android applications [49]. We build upon such work by investigating the challenges that developers face when manually repairing these common issues.

Prior debugging theories break the debugging process into high-level stages [50], [51], [52]. More recent work further describes the actions in the information gathering process of debugging [1], [53]. These processes apply to our results but do not include framework-specific challenges.

VI. Conclusions

We have presented qualitative insights from framework directive debugging scenarios, and the challenges found in framework debugging, such as the difficulty of understanding method call ordering. While these insights and challenges are not necessarily framework specific, they occur while debugging frameworks and are likely more prevalent in framework debugging than other contexts. During this study, we also looked into the difficulty of solving various directive violations by consequence and found one of the most surprising results from our study: that fixing directive violations is more complex than identifying the fault location. Thus, developers need to be both aware of directive violations and possible fixes to quickly address these issues.

Acknowledgment

This material is based upon work supported by AFRL and DARPA under agreement number FA8750-16-2-0042, the National Science Foundation (NSF) under the Graduate Research Fellowship Program (Grant No. DGE1252522) and CCF-1560137 and CCF-1618220; the authors are grateful for their support. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the AFRL, DARPA, NSF, or the U.S. Government.

References

[1] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions of Software Engineering*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

[2] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *Visual Languages - Human Centric Computing*, ser. VLHCC '04, 2004, pp. 199–206.

[3] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, "How Programmers Debug, Revisited: An Information Foraging Theory Perspective," *IEEE Transations of Software Engingeering*, vol. 39, no. 2, pp. 197–215, Feb. 2013.

[4] C. Jaspan and J. Aldrich, "Are object protocols burdensome?: An empirical study of developer forums," in *SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU '11, 2011, pp. 51–56.

[5] R. E. Johnson, "Documenting frameworks using patterns," in *Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '92, 1992, pp. 63–76.

[6] G. Fairbanks, D. Garlan, and W. Scherlis, "Design fragments make using frameworks easier," in *Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06, 2006, pp. 75–88.

[7] J. Ciera, "Proper plugin protocols," Ph.D. dissertation, Carnegie Mellon University, 2011.

[8] U. Dekel and J. D. Herbsleb, "Improving API documentation usability with knowledge pushing," in *International Conference on Software Engineering*, ser. ICSE '09, 2009, pp. 320–330.

[9] R. K. Yin, *Case Study Research: Design and Methods*, 4th ed. Thousand Oaks California, USA: Sage Publications, 2008.

[10] K. Charmaz, *Constructing Grounded Theory*, 2nd ed. Los Angeles, CA: Sage Publications, 2014.

[11] M. Schreier, *Qualitative Content Analysis in Practice*, ser. EBL-Schweitzer. Thousand Oaks California, USA: SAGE Publications, 2012.

[12] C. Larman, *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed. Upper Saddle River, New Jersey, USA: Upper Saddle River, N.J. : Prentice Hall Professional Technical Reference, 2004.

[13] N. E. Beckman, D. Kim, and J. Aldrich, "An empirical study of object protocols in the wild," in *European Conference on Object-Oriented Programming*, ser. ECOOP'11, Berlin, Heidelberg, 2011, pp. 2–26.

[14] Google, "Android," www.android.com, 2017, accessed: 2/15/17.

[15] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[16] U. Dekel, "Increasing awareness of delocalized information to facilitate API usage," Ph.D. dissertation, Carnegie Mellon University, 2009.

[17] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: A critical review and guidelines," in *International Conference on Software Engineering*, ser. ICSE '16, 2016, pp. 120–131.

[18] D. Piorkowski, S. D. Fleming, C. Scaffidi, M. Burnett, I. Kwan, A. Z. Henley, J. Macbeth, C. Hill, and A. Horvath, "To fix or to learn? how production bias affects developers' information foraging during debugging," in *International Conference on Software Maintenance and Evolution*, ser. ICSME '15, Sep. 2015, pp. 11–20.

[19] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon, "Programmers are users too: Human-centered methods for improving programming tools," *Computer*, vol. 49, no. 7, pp. 44–52, July 2016.

[20] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459 – 494, 1985.

[21] C. Jaspan and J. Aldrich, "Checking framework interactions with relationships," in *Proceedinds of the European Conference on Object Oriented Programming*, ser. ECOOP '09, 2009, pp. 27–51.

[22] D. Hou and H. J. Hoover, "Using scl to specify and check design intent in source code," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 404–423, 2006.

[23] C. Jaspan and J. Aldrich, "Retrieving relationships from declarative files," in *Relationships and Associations in Object-Oriented Languages*, ser. RAOOL '09, 2009, pp. 1–4.

[24] W. Wang and M. W. Godfrey, "Detecting API Usage Obstacles: A Study of iOS and Android Developer Questions," in *Mining Software Repositories*, ser. MSR '13, 2013, pp. 61–64.

[25] D. Hou and L. Li, "Obstacles in using frameworks and apis: An exploratory study of programmers' newsgroup discussions," in *International Conference on Program Comprehension*, ser. ICPC '11, June 2011, pp. 91–100.

[26] J. Stylos and B. Myers, "Mapping the space of api design decisions," in *Visual Languages and Human-Centric Computing*, ser. VL/HCC '07', 2007, pp. 50–60.

[27] B. Ellis, J. Stylos, and B. Myers, "The factory pattern in api design: A usability evaluation," in *International Conference on Software Engineering*, ser. ICSE '07, 2007, pp. 302–312.

[28] J. Stylos and S. Clarke, "Usability implications of requiring parameters in objects' constructors," in *International Conference on Software Engineering*, ser. ICSE '07, 2007, pp. 529–539.

[29] J. Stylos and B. A. Myers, "The implications of method placement on api learnability," in *SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT/FSE '08, 2008, pp. 105–112.

[30] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *International Conference on Software Maintenance*, ser. ICSM '13, 2013, pp. 70–79.

[31] G. Bavota, M. Linares-Vásquez, C. E. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "The impact of api change- and fault-proneness on the user ratings of android apps," *IEEE Transactions of Software Engineering*, vol. 41, no. 4, 2015.

[32] W. Wu, F. Khomh, B. Adams, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of api changes and usages based on apache and eclipse ecosystems," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2366–2412, Dec 2016.

[33] M. Borg, E. Alégroth, and P. Runeson, "Software engineers' information seeking behavior in change impact analysis: An interview study," in *International Conference on Program Comprehension*, ser. ICPC '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 12–22.

[34] C. W. Choo, B. Detlor, and D. Turnbull, "Information seeking on the web: An integrated model of browsing and searching," *First Monday*, vol. 5, no. 2, Feb. 2000.

[35] J. Lawrance, R. Bellamy, and M. Burnett, "Scents in programs: Does information foraging theory apply to program maintenance?" in *Visual Languages and Human-Centric Computing*, ser. VL/HCC '07, 2007, pp. 15–22.

[36] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "An empirical investigation on documentation usage patterns in maintenance tasks," in *International Conference on Software Maintenance*, ser. ICSM '13, 2013, pp. 210–219.

[37] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, and A. Bacchelli, "Information needs in contemporary code review," *Human-Computer Interaction*, vol. 2, no. CSCW, pp. 135:1–135:27, Nov. 2018.

[38] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, "What should developers be aware of? An empirical study on the directives of API documentation," *Empirical Software Engineering*, vol. 17, no. 6, pp. 703–737, 2012.

[39] M. Bruch, M. Mezini, and M. Monperrus, "Mining subclassing directives to improve framework reuse," in *Mining Software Repositories*, ser. MSR '10, 2010, pp. 141–150.

[40] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing apis documentation and code to detect directive defects," in *International Conference on Software Engineering*, ser. ICSE '17, 2017, pp. 27–37.

[41] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and Z. Xuejiao, "Improving api caveats accessibility by mining api caveats knowledge graph," 09 2018, pp. 183–193.

[42] T. D. LaToza and B. A. Myers, "Hard-to-Answer Questions About Code," in *Evaluation and Usability of Programming Languages and Tools*, 2010, pp. 1–6.

[43] A. J. Ko and B. A. Myers, "Debugging reinvented: Asking and answering why and why not questions about program behavior," in *International Conference on Software Engineering*, ser. ICSE '08, 2008, pp. 301–310.

[44] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *Foundations of Software Engineering*, ser. ESEC/FSE '17, 2017, pp. 117–128.

[45] T. Kulesza, S. Stumpf, M. Burnett, W. K. Wong, Y. Riche, T. Moore, I. Oberst, A. Shinsel, and K. McIntosh, "Explanatory debugging: Supporting end-user debugging of machine-learned programs," in *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2010, pp. 41–48.

[46] C. Lebeuf, E. Voyloshnikova, K. Herzig, and M.-A. Storey, "Understanding, debugging, and optimizing distributed software builds: A design study," in *International Conference on Software Maintenance and Evolution*, ser. ICSME '18, 2018, pp. 496–507.

[47] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design space of bug fixes and how developers navigate it," *IEEE Transactions on Software Engineering*, vol. 41, no. 1, pp. 65–81, Jan 2015.

[48] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 408–419.

[49] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 187–198.

[50] J. D. Gould, "Some Psychological Evidence on How People Debug Computer Programs," *International Journal of Man-Machine Studies*, vol. 7, no. 2, pp. 151 – 182, 1975.

[51] I. R. Katz and J. R. Anderson, "Debugging: An analysis of bug-location strategies," *Human-Computer Interaction*, vol. 3, no. 4, pp. 351–399, Dec. 1987.

[52] V. Grigoreanu, M. Burnett, S. Wiedenbeck, J. Cao, K. Rector, and I. Kwan, "End-User debugging strategies: A sensemaking perspective," *ACM Transactions on Computer-Human Interaction*, vol. 19, no. 1, pp. 1–28, 2012.

[53] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers, "Program comprehension as fact finding," in *Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07, New York, NY, USA, 2007, pp. 361–370.