UNIVERSITY OF CALIFORNIA,
IRVINE

Toward A Novel Tool for Incorporating Video Into Effective Bug Reporting

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Nadia Etemadi

Thesis Committee:
Professor André van der Hoek, Chair
Professor David Redmiles
Associate Professor James A. Jones

2023

DEDICATION

To my parents, who celebrate me in my highest highs and uplift me in my lowest lows.

I am only one, but I am one. I cannot do everything, but I can do something. And because I cannot do everything, I will not refuse to do the something that I can do.
– Edward Everett Hale

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

ACKNOWLEDGMENTS **vi**

ABSTRACT OF THE THESIS **vii**

1 Introduction **1**

2 Background **5**
2.1 The Bug Reporting Process 5
2.2 Existing Bug Reporting Tools 7
2.3 Ease of Access to Bug Reporting 8
2.4 Resolution Time Reduction 9
2.5 Bug Reports with Video 10

3 VideoLab **12**
3.1 Objective 12
3.2 High-Level Design 13
3.2.1 Video Uploading 13
3.2.2 Domain-Specific Annotations 14
3.2.3 Simplified Video Editing 15
3.3 User Interface 16
3.3.1 The Video Editor 16
3.3.2 Text and Shape Annotations 16
3.3.3 A Visually Non-Complex Interface 18

4 VideoLab in Practice **20**
4.1 BugZilla Report 1560660 20
4.2 BugZilla Report 1546326 21
4.3 BugZilla Report 1028819 26

5 Evaluation Plan **28**
5.1 Questions for Assessment 28
5.2 Evaluation Stages 30
5.2.1 Phase I: Design Critique and Usability Evaluation 30

5.2.2 Phase II, Video Creation 32
5.2.3 Phase II, Video Consumption 33

6 Useful Future Extensions **36**
6.1 In-App Screen Recording 36
6.2 Voiceovers 37
6.3 Video Blurring 38
6.4 Steps Recorder Integration 38
6.5 Selecting Annotation Duration 39
6.6 Video Editing Timeline 40
6.7 Bug Report Exporting and Submission 40

7 Conclusion **42**

Bibliography **43**

LIST OF FIGURES

Page

2.1 BugZilla report 1798314: Firefox freezes when used with NVDA screen reader 6

3.1 VideoLab Video Upload 14
3.2 VideoLab Editing Interface 17
3.3 Shape Annotations 17
3.4 Text Annotations 17
3.5 Trimming 18
3.6 Dialogue Box with Annotation Duration 19

4.1 BugZilla report 1560660 reproduction steps 21
4.2 BugZilla report 1560660 video attachment 22
4.3 BugZilla report 1560660 video before editing 22
4.4 BugZilla report 1560660 video after editing 23
4.5 BugZilla report 1546326 reproduction steps 24
4.6 BugZilla report 1546326 video before editing: 16s mark 24
4.7 BugZilla report 1546326 video before editing: 48s mark 25
4.8 BugZilla report 1546326 video after editing: 16s mark 25
4.9 BugZilla report 1546326 video after editing: 48s mark 26
4.10 BugZilla report 1028819 reproduction steps 27
4.11 BugZilla report 1028819 video after editing 27

5.1 Experiment Design 31
5.2 Video Creation Experimental Measures 33

ACKNOWLEDGMENTS

I would like to extend my deepest gratitude to my advisor, Professor André van der Hoek. His support and guidance through these past two years have been invaluable in not only the completion of my degree, but also my personal growth as an academic and a software developer.

I would also like to thank the members of the Software Design and Collaboration Laboratory for supporting me through this process and providing support, feedback, and a general sense of community during my time here at UC Irvine. In particular, I'm most grateful to my fellow graduate Elahe Paikari, for working so closely with me and allowing me to expand upon her research for this thesis.

I would finally like to thank Professor David Redmiles and Professor James Jones for providing invaluable feedback and asking challenging, thought-provoking questions. I am incredibly grateful to have worked with and gotten to know them and the other Software Engineering professors during my time here at UC Irvine.

ABSTRACT OF THE THESIS

Toward A Novel Tool for Incorporating Video Into Effective Bug Reporting

By

Nadia Etemadi

Master of Science in Software Engineering

University of California, Irvine, 2023

Professor André van der Hoek, Chair

Management of bug reports is a necessary evil in the software development process. Report management involves handling user-submitted bug reports, which is known to be a time-consuming task because the developers need to sift through many reports, with many containing insufficient information. This lack of information is partially attributed to a gap in knowledge: users without relevant technical knowledge often do not know how to appropriately report a bug. Attaching a video recording to such reports has shown promise in reducing bug resolution time in bug reporting for mobile applications, and might have the potential for similar results for other types of systems. However, editing or even creating videos specifically for bug reporting is not a simple task. In this thesis, I introduce VideoLab, a novel prototype tool to simplify the creation of videos for bug reports. VideoLab is based primarily around a pared-down video editing interface with a set of dedicated annotations that aim to make communicating defective behavior to developers easier for end-users.

Chapter 1

Introduction

Debugging code is an inevitable activity in the process of software development. So inevitable, in fact, that the average industry developer spends nearly half of their programming time debugging their code [7]. Clearly, a decrease in time spent debugging would open up time that developers can spend on other activities to further their products and projects. Developer productivity and efficiency is not only in the best interest of the developers themselves, but also of the company they work for, since optimizing developer productivity can significantly reduce cost at a high level [7]. Nearly half of every dollar spent on software as an industry goes to debugging, making debugging an extremely expensive and inefficient process with regard to creating and shipping software [2]. This reinforces the notion that debugging is integral to the software development workflow, for better or for worse.

A number of factors have the potential to affect developer productivity surrounding the debugging process. Poor quality of the existing code and lack of innovative infrastructure tools are negatively linked to developer productivity [8]. With developers making extensive use of debugging tools to assess the behavior of their code [4], this could suggest that any further advancements to debugging tools could greatly benefit developer productivity by

allowing them locate issues more quickly. By these metrics, a productive developer is one who writes high-quality code without needing to spend time combing it for bugs, as well as one who has access to quality debugging tools to achieve the goal of well-crafted code.

However, some developers have to deal with more than just bugs they encounter themselves. A portion of bugs handled by developers includes bugs submitted by end-users of the product developers create—especially if the software functions in a largely user-facing capacity. In the case of open-source projects, a non-trivial portion of user-submitted bug reports end with a developer explaining how to achieve the desired behavior with existing features, effectively acting as technical support rather than debugging [13]. Filtering through bug reports raises the question of how users can create informative, useful bug reports to ease the burden on developers. The disparity in quality of bug reports that lack important information is due to mismatched expectations in what developers require and what users typically provide. When reporting a bug, users may provide what they believe to be important information, but often that information is not what developers need to diagnose and fix the bug [5]. This is especially true when users lack software development experience or knowledge of the bug reporting process—someone who has never been in an environment like that may not know about steps to reproduce, for example.

Video recording as a tool to communicate information about technology is well-documented [3, 9, 10, 22]. Researchers in the field of mobile applications have observed mobile devices and applications with extensive use of video recording, for instance, to conduct usability testing, either by screen recording an application or recording the physical device [22]. Because mobile applications have numerous digital and physical components that could be difficult to describe in a principally written format, a visual account of the application's behavior allows users to demonstrate the behavior as it occurs without the limitation of written descriptions. Despite the clear value in this and other domains [14, 15], the use of video remains largely unexplored with regard to communicating information in bug reporting.

The ability to communicate effectively with video can serve to potentially lessen the need for end-users to have specialized knowledge or experience concerning bug reporting.

The inclusion of video in a bug report can positively impact a bug report's resolution time [5]. However, not all videos are created equally. In an effort to enhance the bug reporting process, I have designed VideoLab, a novel prototype for capturing and editing recordings of applications experiencing defective behavior. VideoLab is a web application designed to provide a lightweight, easily understood method of editing videos that are specifically intended to be attached to bug reports, making it more accessible to end-users. The app particularly allows a user to edit video recordings by adding text and shape annotations that are particularly helpful to highlight defective behavior and export them.

VideoLab simplifies the video creation process while still ensuring key elements are present in the video. VideoLab's pared down video editing interface provides a dedicated set of annotations allowing a user reporting a bug to describe exactly what is happening, and where in the video it is happening. In this manner, an end-user creating a video can describe reproduction steps in a visual format alongside a written report. A simplified but specialized set of annotations can convey information about the behavior in the video efficiently [3], more so than with text. These features aim to move the bug resolution process along in a more efficient manner by simplifying the information a reporter needs to provide, together with allowing the reporter to provide it in a way that is easy to understand regardless of reporting experience.

This thesis contributes a high-level description of VideoLab as a novel prototype, as well as a detailed plan of evaluation for VideoLab's capabilities. It also describes how VideoLab aims to improve upon the bug resolution process by allowing for an easier, more streamlined video attachment creation process.

The remainder of this thesis is organized as follows. Chapter 2 provides relevant information

about reducing bug resolution time, context about bug reports with video included, accessibility of bug reporting to end-users of varying experience, and a summary of current bug reporting tools that attempt to remedy the issues described above. Chapter 3 details the primary objective of VideoLab and the design choices made in developing the tool, explaining the choices at a high level and summarizing the user interfaces. Chapter 4 presents an informal illustration of VideoLab in use with three different sample bug reports. Chapter 5 lays out a plan for evaluating the tool, including assessment questions and stages of evaluation. Chapter 6 details future directions for development of the tool. Chapter 7 wraps up with a conclusion.

Chapter 2

Background

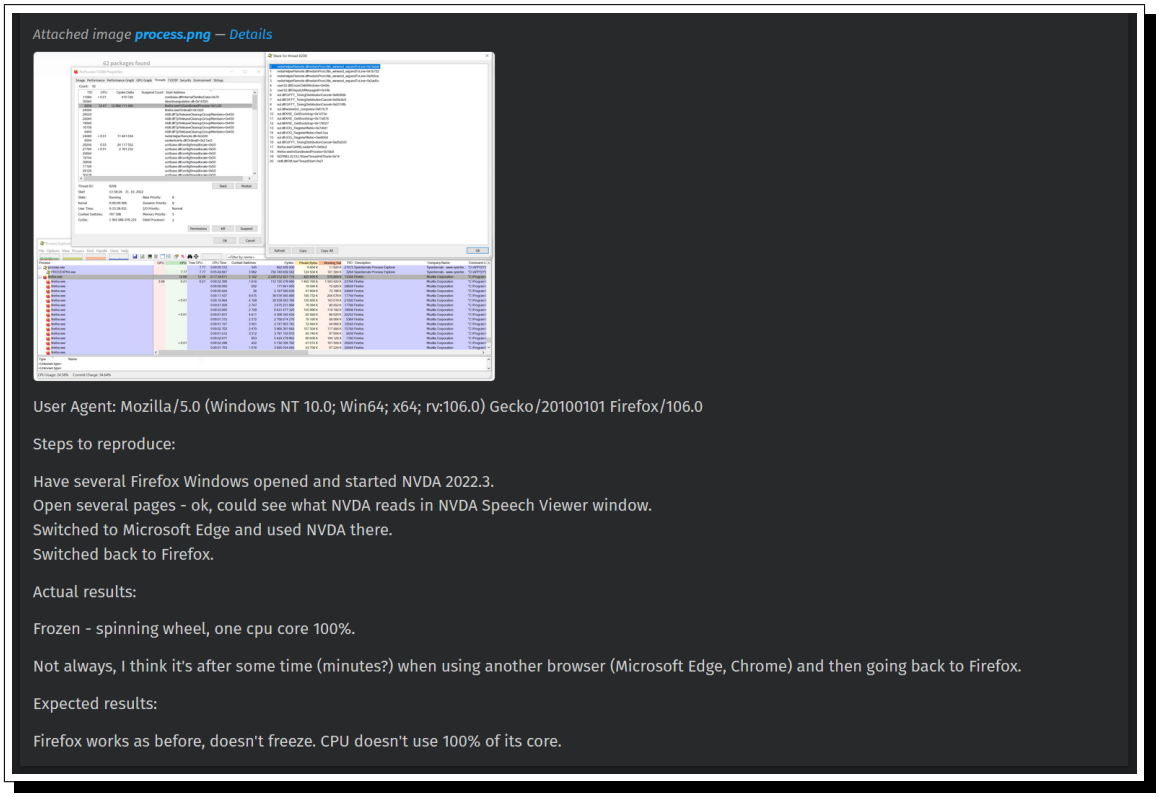
This chapter presents relevant background material on the bug reporting process, existing bug reporting tools, ease of access to bug reporting for end-users, research in reduction of bug resolution time, and video attachments on bug reports.

2.1 The Bug Reporting Process

When a user encounters a defect in an application they are using, they can choose to send a bug report to the developers of the project. A typical bug report provides a variety of fields for the reporter to fill out with relevant information to communicate to the developer what happened. These fields include a title, summary of behavior, expected versus actual behavior, and information about the user's device. An example bug report from Mozilla's BugZilla bug tracking system[1] is shown in Figure 2.1. In it, a user reports a bug where Firefox freezes after using a screen reader in a Firefox window, using the same screen reader in a Microsoft Edge window, and returning to Firefox.

[1] https://bugzilla.mozilla.org/home

Figure 2.1: BugZilla report 1798314: Firefox freezes when used with NVDA screen reader



Upon receipt of a bug report, developers will triage the bug, which typically first involves investigating if it is a duplicate and, if so, marking it as such. If it is not a duplicate, the developer will interpret the report more closely and still may reject it for a variety of reasons, such as incorrect product. An inability to reproduce the behavior described in the bug report and incomplete information are two common reasons why reports may be rejected [25]. If a bug report passes the triage process, it is assigned to a developer to look into it further, normally by investigating aspects such as the severity of the bug, categorizing the bug, and setting up their environment to begin resolving the defective behavior [25]. Sometimes a somewhat different process is followed, with the triager assessing the severity and categorizing the bug before a developer is assigned. Regardless, the nature of the bug reported has to be understood.

2.2 Existing Bug Reporting Tools

Currently there are a number of general bug reporting tools that allow developers and end-users to communicate, with bug trackers such as GitHub Issues[2] and JIRA[3] being quite popular. Sometimes bug reporting interfaces are built directly into web apps, such as within the browsers Google Chrome[4] and Mozilla Firefox[5]. Still other applications have bug reporting capabilities built into them, such as note-taking app Notability[6], which allows users to report an issue by filling out a form from within the software. Most of these directly are functional, but lack useful guidance on bug reporting and feedback for users on the reports they create [24].

There have been several attempts to mitigate this issue, with varying degrees of success. One such attempt is CueZilla, a tool that measures the quality of bug reports and recommends improvements based on its contents, with an example suggestion being to provide additional information when the report is thin in its content [5]. Another tool is Fusion, a web-based interface for bug reporting that allows users a more interactive reporting experience. Rather than typing, they can select, drag, and drop elements that represent the components of their bug report [16]. A more recent example is EBug, which is a mobile bug reporting system similar to Fusion. It goes one step further and suggests appropriate reproduction steps by predicting what steps follow as the user is providing them [10].

Though these tools provide meaningful steps toward a bug reporting experience that reduces the knowledge needed by a reporter to create an informative report, many of them have various shortcomings that prevent them from addressing the issues with bug reporting in their entirety. For instance, Fusion's design was not conducive to end-user use, which

[2]https://docs.github.com/en/issues
[3]https://www.atlassian.com/software/jira
[4]https://www.google.com/chrome/index.html
[5]https://www.mozilla.org/en-US/firefox/new/
[6]https://notability.com/

made it *more* difficult for them to report bugs. Inexperienced users using FUSION found difficulty with a complex interface and lack of guidance on creating detailed bug reports, tending mostly to create superficial reports instead [16]. Previous systems also had a lack of interactivity with regard to the feedback given. BURT, an interactive bug reporting chatbot developed by Song *et. al*, represents a more substantial approach toward mitigating issues with current bug reporting tools [24]. BURT does take a significant step toward an easier, more novel bug reporting experience by providing a chatbot-style interface for users to interact with. It also predicts potential next reproduction steps as the user enters their own, effectively acting as a guide walking the end-user through the reporting process. However, despite its strengths, BURT's current handling of different bug scenarios is limited primarily to GUI-based issues, and it was not evaluated in terms of how developers handled the bug reports it produced [24].

2.3 Ease of Access to Bug Reporting

Management of bug reports is integral to maintaining a smooth workflow in a development team and is relatively costly to the software development industry. Most bugs are manually reported, as only certain types can be reported automatically, such as an application crashing [24]. As previously mentioned, a significant discrepancy in knowledge exists between users and developers, more specifically with regard to what users know and what developers need [5]. While many systems exist to facilitate manual bug reporting and ease the process for users in an attempt to bridge this gap in knowledge, these systems primarily offer limited guidance on the necessary information for the report and how to properly include it [24]. Manual bug reporting remains relatively inaccessible to end-users despite many attempts to mitigate it, such as Mozilla's bug report writing guidelines[7]. This not only puts a sizeable burden on developers to find and fix these bugs themselves, but it also may result in user

[7]https://bugzilla.mozilla.org/page.cgi?id=bug-writing.html

dissatisfaction with the product in question—if an application is buggy or crashes constantly, a user expends less effort by simply not using it instead of submitting a bug report that may not be addressed anyway. These observations suggest that bug reporting presents various usability barriers to users, and adjustments to the current bug reporting methodologies are necessary in order to bridge these barriers.

2.4 Resolution Time Reduction

When developers are tasked with resolving a bug, the developers often experience delays in beginning the resolution process after a bug assignment. This delay stems from a number of factors, primarily related to the need to more fully understand the bug. This includes activities such as performing an initial investigation, verifying whether the bug is an actual bug, interpreting the description of the bug, and exchanging comments with the team. Severity of the bug can significantly cut down on the delay before a developer begins fixing the bug [27]. For instance, a critical bug requires an imminent fix so as to reduce major damage or impact on the codebase.

Pertaining to the bug report itself, various elements of the report carry significance and can impact resolution time based on whether or not they are included, as well as the quality of each element. Generally, developers will first try to reproduce the behavior described in the report. More complex bugs can also benefit from inclusion of a stack trace or crash dump to aid in reproduction. Reproduction steps have been found to be the most important to include when it comes to thinking about what to include in a bug report [23]. Another desirable feature of bug reports that can impact resolution time is a description of expected versus actual behavior of a defective application. A description of what the end-user expected to see versus what they actually saw can aid the developer in identifying the source of the bug, as well as assessing whether something is a bug or an intentional feature in the first place.

Additional components of a bug report outside of the developer's control can also impact resolution time, such as commenting activity from the reporter or other developers, the product itself, the component experiencing issues, and the product's version [19]. The language of the codebase itself can also impact resolution time–for instance, Java projects generally exhibit faster resolution time than Ruby projects [28]. Ostensibly, there is only so much of the debugging process that is within the developer's control, which in turn necessitates some knowledge and guidance on the part of the user reporting the bug to do so in a manner that helps the developer as much as possible.

2.5 Bug Reports with Video

Modern user-facing applications experience a majority of their bugs in the graphical user interface (GUI), so much so that many Android and iOS mobile apps have built-in screen recording capability to facilitate capturing the behavior causing an issue [9]. Despite the steadily growing use of images and video to provide context to bug reports, such as on the discussion platform StackOverflow [18], adding video to bug reports does pose a number of challenges for the developers tasked with handling the reports. The main issue with video attachments is the added difficulty of determining if two videos represent duplicate bugs. At present, written bug reports can easily be marked as duplicates [6, 20], but video reports collected at scale can present a sizeable challenge for developers. Tools such as Tango [9] have been developed to mitigate the issue of identifying duplicate video reports, allowing video attachments to reports to slowly but surely make their way into the software development process.

Videos attached to bug reports also provide benefits to non-technical users who may struggle to document the bug along with reproduction steps. Even the most seasoned developers may potentially struggle with describing a bug, so describing defective behavior in an applica-

tion could present a much larger challenge to someone with no technical knowledge. Video recording provides more detail and context than a written description may be able to, and can potentially engage users in wanting to provide more feedback [11]. Attaching video to bug reports provides reporters with greater investment in the process while also providing developers with the bug's information in a more concise, digestible format.

Chapter 3

VideoLab

This chapter introduces VideoLab, a novel prototype of a video editing tool that serves to enable end-users to create video attachments to supplement their bug reports. This chapter presents the tool's objective, the high-level design decisions and how they align with the objective, and a description of the main interfaces present in the application.

3.1 Objective

VideoLab is a novel prototype that aims to simplify the process of creating video attachments for bug reports, regardless of amount of experience. The principal objective of VideoLab is to **ease communication of defective behavior to developers through the use of dedicated video annotations.**

VideoLab's primary goal is to give end-users the ability to communicate more easily without needing the precise technical knowledge to describe the behavior they see in a defective application. Studies in the field of mobile application usability testing suggest that using video to communicate information is more efficient than doing so verbally or through text [3].

VideoLab aims to facilitate end-users creating videos that clearly display the issues they are experiencing by providing domain-specific annotations. In addition to reducing the amount of decision-making necessary on how to annotate the video, the dedicated set of annotations provided in the editing interface allow users to create focus around different areas in the video and add clarifying text if desired when the visuals presented in the video are unclear.

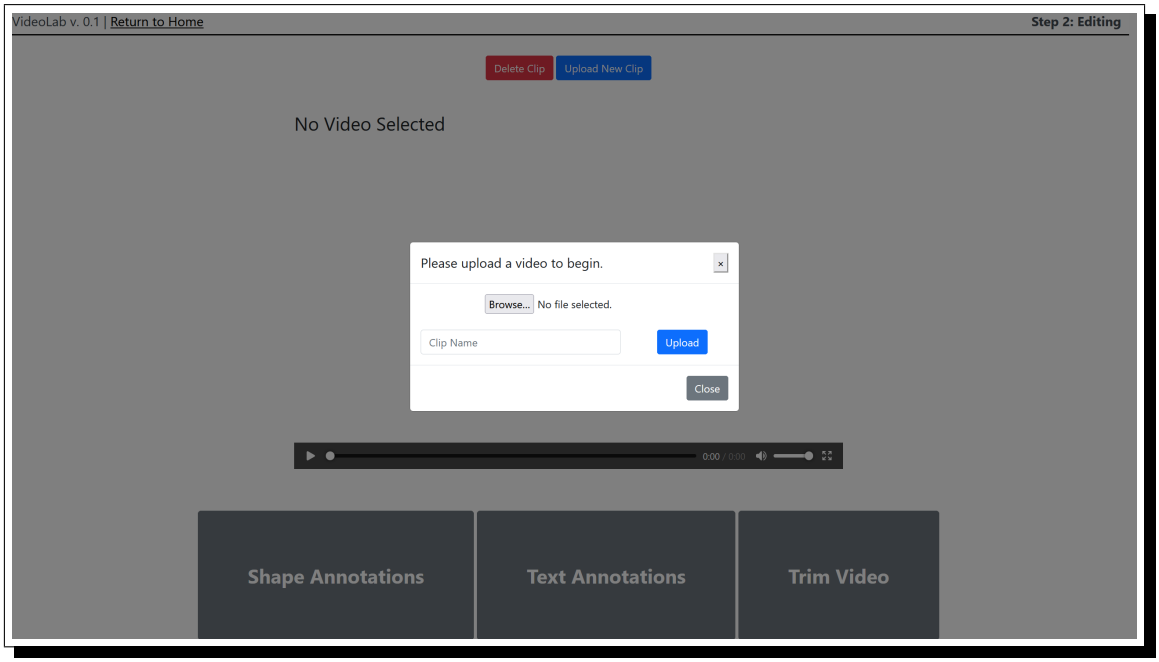
3.2 High-Level Design

The primary design decisions made in the creation of VideoLab center around a simple, visually non-complex interface and a set of specifically curated video editing tools.

3.2.1 Video Uploading

In order to begin the editing process, users need to upload a previously recorded video clip of the defective behavior they saw in an application. When the page first loads, a modal appears prompting the user to upload a video, as shown in Figure 3.1. The upload process is fairly straightforward—the user clicks the "Browse" button in the modal, selects their video clip from the file explorer, and clicks "Upload". Upon uploading, the video appears in a preview pane in the center of the page, above a set of three buttons designating different editing capabilities. Some particular design choices I made in this area were ensuring the prominence of the video preview pane in the center of the page and the editing toolbar buttons at the bottom.

Figure 3.1: VideoLab Video Upload



3.2.2 Domain-Specific Annotations

The annotations present in the VideoLab editing toolbar are curated with the intention of allowing end-users to clearly display in their video clip the defective behavior they are witnessing. The shapes present in the annotation palette are brightly colored with thick lines, ensuring visibility in a manner similar to the large type and spacing of the text on the interface and the bright colors of the buttons. In particular, each type of annotation provided serves a specific purpose. Circles of two different sizes are provided to allow users to place them on the video clip and indicate focus on either a somewhat large or smaller area. Arrows pointing upward to the left and upward to the right are provided to signal a more specific focus on a particular element of the video if the small circle is not sufficient. A translucent yellow box is also provided to allow the user to highlight and draw visual attention to a specific area of the video. If the user wishes to communicate information in a written format on the video, they also have the option to generate and include short lines of text directly on the video clip.

3.2.3 Simplified Video Editing

Rather than provide the user with a whole suite of editing capabilities, I have identified and provided the most crucial ones: trimming and annotation overlaying. Due to the overall complexity of video as a medium [12], video editing can be an incredibly complex process. In keeping with the objective of allowing the user to clearly communicate information about defective behavior with annotations, I chose only two video editing functions because I felt that other functions normally included in video editing programs, such as visual effects, would be unnecessary.

Allowing the user to trim the video is important because this function allows the user to remove parts of the video that are irrelevant to the defective behavior being showcased, which contributes to the goal of facilitating clear communication between end-user and developer. The trimming functionality requires very little effort on the part of the user—the interface asks for a start and end point, performs the action with a button click, and renders the resulting clip for the user.

Overlaying text and image annotations on top of the video clip is another function I wanted to include, as I felt it would allow the end-user to provide additional visual information to the developer through the video clip. As previously stated, visual information has the potential to convey information in a more easily understood manner than written information, and providing dedicated annotations allows users to take further advantage of communicating visually rather than textually. The editing palette has a collection of shapes that users can drag and drop on top of the clip, and a dialogue box appears prompting the user to choose the duration that the shape will appear on the clip, as well as the ability to specify where in the clip the shape should begin to appear. Users also have the ability to generate customized text to overlay on the video. The simplified, dedicated collection of shapes and the ability to add text to the video clip contributes to VideoLab's goal of facilitating communication

between end-users and developers through the use of annotations.

3.3 User Interface

VideoLab's main interface is the video editing screen. VideoLab is a web application intended for desktop use. Figure 3.2 shows the video editing interface after a video has been uploaded.

3.3.1 The Video Editor

Once the user uploads a video recording they have created, they are taken to the main video editing page, as shown in Figure 3.1. The user sees the clip preview in the center of the page, along with three large buttons for selecting specific editing features. Figures 3.3, 3.4, and 3.5 show each of the editing toolbar buttons in more detail respectively. This toolbar provides the end-user with the ability to trim the video, generate customized text to be rendered on the video, and drag and drop shapes to be rendered on top of the video.

3.3.2 Text and Shape Annotations

When the user places the chosen annotation on the video editing area, a dialogue box—shown in Figure 3.6—appears prompting the user to enter a starting time stamp for where the annotation should appear, as well as a set of buttons to choose how long the annotation should appear on the video. In accordance with VideoLab's goal of focusing on the user's ability to communicate information through the video clip, the buttons show users preset durations—one second, five seconds, ten seconds, and the entire video. The dialogue box is structured in this way to minimize effort on the part of the user and maintain focus on conveying information through the video clip.

Figure 3.2: VideoLab Editing Interface

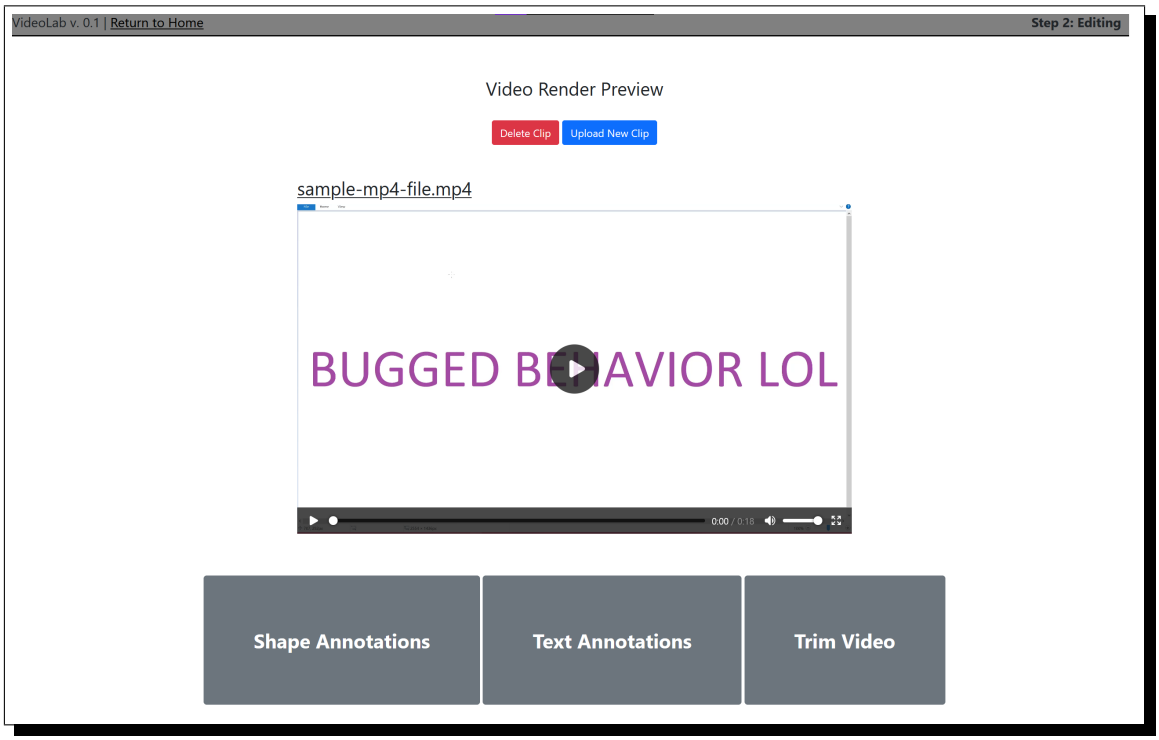


Figure 3.3: Shape Annotations

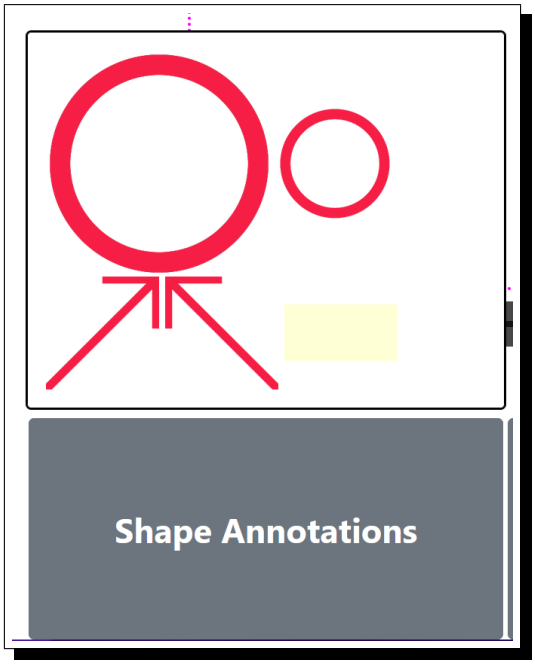


Figure 3.4: Text Annotations

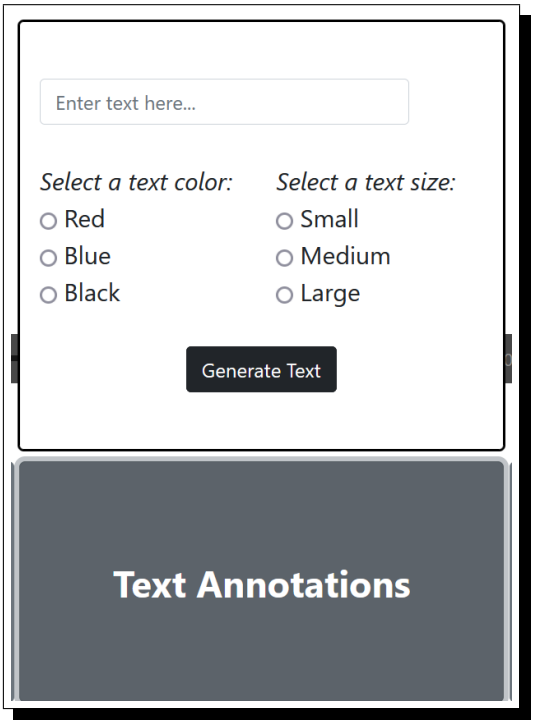
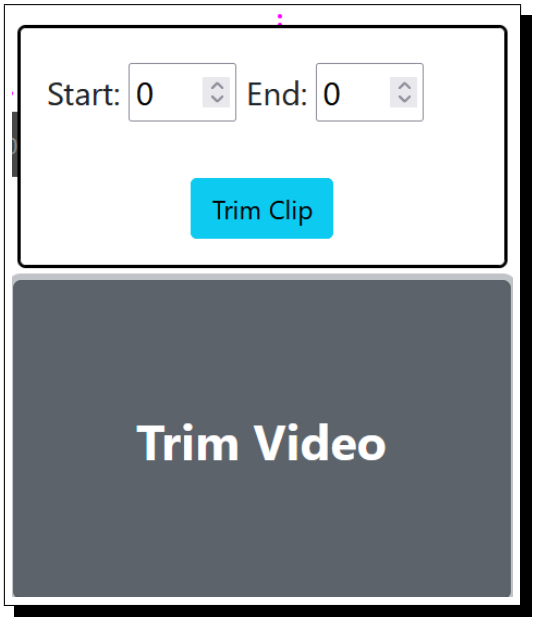


Figure 3.5: Trimming



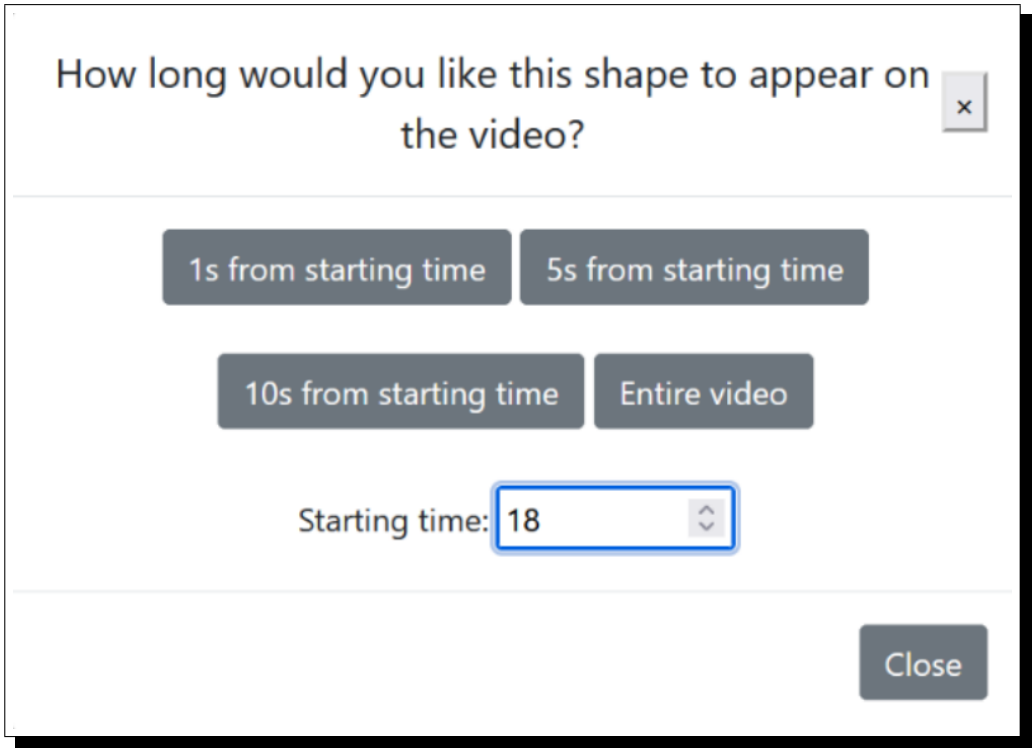
Choosing a duration in the dialogue box prompts VideoLab to render the video with the chosen annotation on top. After rendering the clip with the desired annotation, the video saves to the user's local directory, and it appears on the video preview pane. Once all annotations have been made and the video is rendered, the user can attach it to the bug report they are submitting within their application.

3.3.3 A Visually Non-Complex Interface

The visual complexity of an interface has a great impact on the user's desire to interact with it. Interfaces that have moderate visual complexity are seen by users as more pleasant to use [26]. As a result, maintaining a balance in visual complexity was the main driving factor in designing the user interface of VideoLab.

VideoLab's main interfaces have white backgrounds with a gray bar at the top of the page to indicate the interface the user is seeing. Using neutral, muted colors for background elements eliminates visual distraction, as opposed to bright colors that, in addition to increasing

Figure 3.6: Dialogue Box with Annotation Duration



distraction and complexity, may make the interface difficult to see and read. The interfaces also use Helvetica Neue, which is the default font of Bootstrap[1], the frontend framework used to build VideoLab. Large type and adequate spacing is used where possible to ensure readability of the text, contributing to decreased visual complexity of the interface [21].

The buttons used throughout the interface display large, spaced out text for readability, and provide contrast to the background with bright colors. The toolbar buttons themselves are large with bold text, and the buttons within each editing menu are brightly colored to provide contrast against the neutrally-colored background elements.

[1]http://getbootstrap.com

Chapter 4

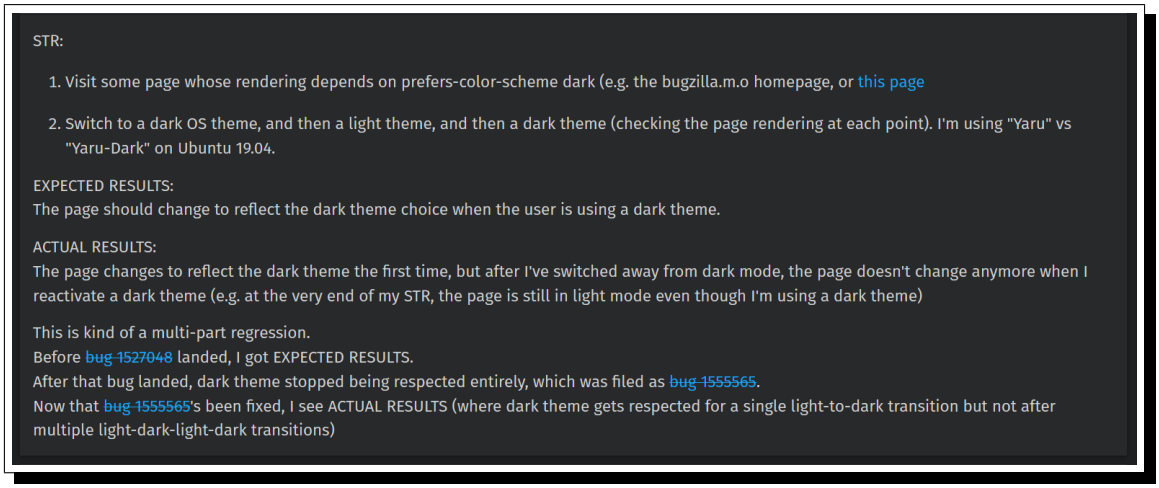
VideoLab in Practice

In this chapter, I present three different bug reports from Mozilla's BugZilla bug tracker, all originally submitted with video. With each bug report, I provide an example of how the videos attached can be edited to enhance the information provided in the bug report.

4.1 BugZilla Report 1560660

Bug report 1560660, titled "prefers-color-scheme media query doesn't reflect user's preference, after the theme has changed away from dark", describes an issue with the Mozilla Firefox browser running on the Ubuntu operating system. When the user swaps between light and dark OS themes, the CSS attribute `prefers-color-scheme` is `light` even when `dark` is selected. The reporter provided steps to reproduce, which are shown in Figure 4.1 as STR. A screenshot of their video attachment along with a short description is shown in Figure 4.2. I chose this bug as an example for two reasons: the defective behavior is present in Firefox's UI, which allows for easy visual comprehension of what is occurring, and the report was submitted by a Mozilla engineer. Though the report is well-written and infor-

Figure 4.1: BugZilla report 1560660 reproduction steps



mative, even small annotations made to the video can enhance the potential for informative communication.

Specifically, the video attachment could benefit from some additional visual cues to illustrate the behavior. I had to watch it a few times to discern the defective behavior. Figures 4.3 and 4.4 show side-by-side comparisons of the original video and the video after being edited with VideoLab, both at the ten-second mark. In Figure 4.4, I placed a highlight over the `light` attribute in the browser, and a small circle around the Yaru-dark theme selection. Though the reporter draws attention to these sections by circling the mouse pointer around each area, using these shapes to draw specific attention to the difference between the `prefers-color-scheme` attribute and the theme selection is preferred, especially since multiple areas of the video can be highlighted at the same time.

4.2 BugZilla Report 1546326

Bug report 1546326, titled "[Dedicated Profiles] The about:profiles page does not display correct information if a profile folder is deleted", details a bug with the Firefox profile feature—if a profile is deleted, the `about:profiles` page in the browser displays incorrect

Figure 4.2: BugZilla report 1560660 video attachment

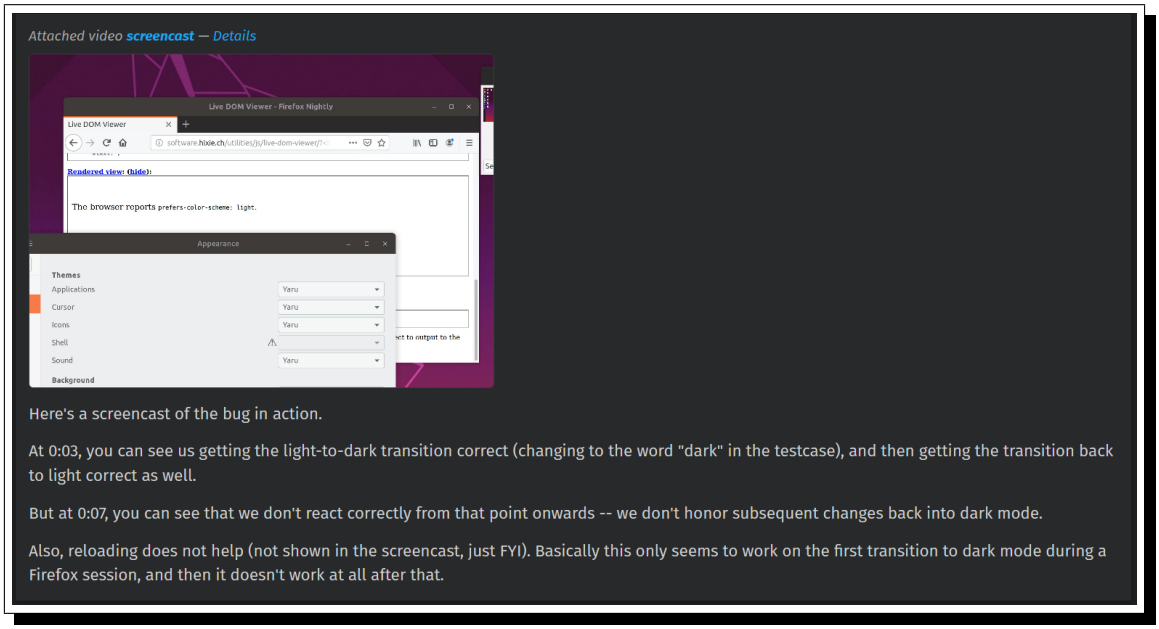


Figure 4.3: BugZilla report 1560660 video before editing

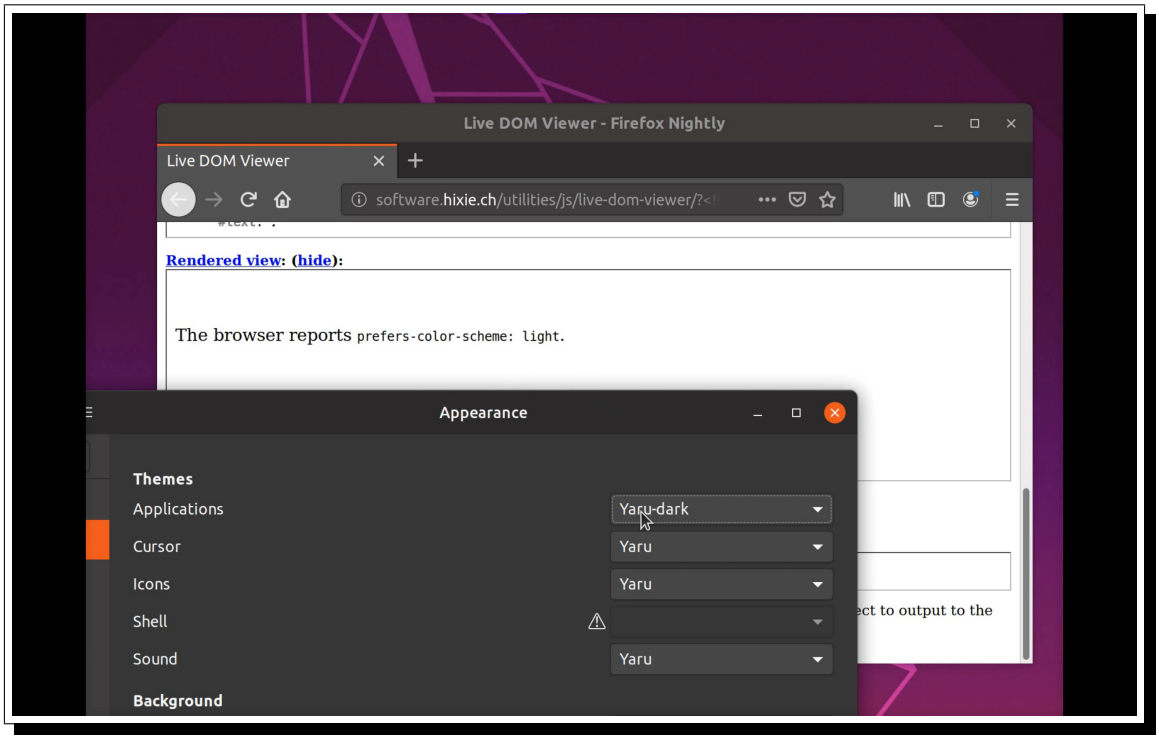
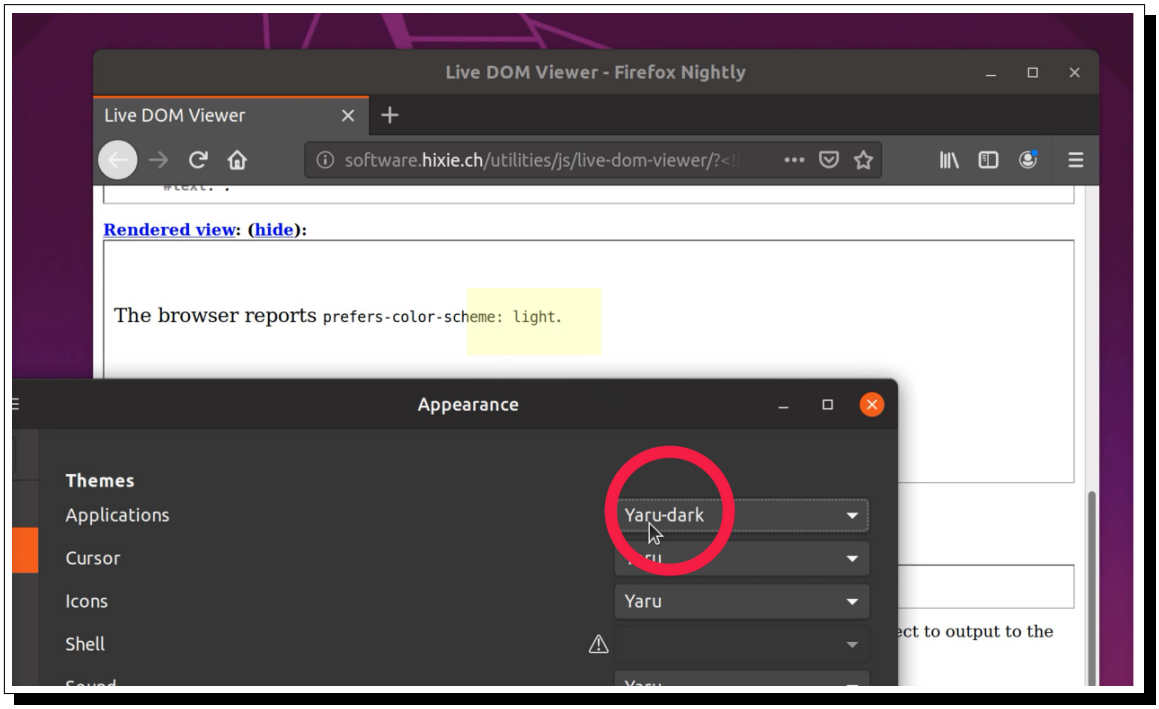


Figure 4.4: BugZilla report 1560660 video after editing



information. The reporter provides steps to reproduce the behavior along with expected and actual results—these are already shown prior in Figure 4.1. The reporter also goes so far as to include additional information about affected versions, the platform in use, and additional notes about the bug. I chose this bug report because it provides adequate written information, and is not UI-related—I want to show how VideoLab's editing capability can be useful for a bug that is not necessarily as visual as the previous example.

Figures 4.6 and 4.7 show the video attached to this report at 16 seconds, when a profile folder is deleted, and at 48 seconds, when the browser is opened to show the incorrect information. Using VideoLab, I have added captions to each frame, shown in Figures 4.8 and 4.9. Though context is provided in the written bug report, adding text captions to the video directly enables the developers addressing this bug to pinpoint the key defective behaviors in the video. In this case, the first annotation summarizes an action in context, and the second pinpoints the problematic result.

Figure 4.5: BugZilla report 1546326 reproduction steps

[Affected versions]:

- Fx67.0b13
- Fx68.0a1

[Affected platforms]:

- Windows 10 x64

[Steps to reproduce]:

1. Launch Firefox and check the about:profiles page. (default-beta profile should be in use).
2. Close Firefox.
3. Go to C:\Users\current_user\AppData\Roaming\Mozilla\Firefox\Profiles and delete the 'default-beta' profile folder.
4. Create a copy of the Firefox install and launch Firefox from that copied folder.
5. Go to about:profiles.

[Expected result]:

- The 'default-beta' profile is still displayed
- The 'Open Folder' buttons are missing for the location from where the profile was deleted (in this case the root path)
- The profile does not appear to be in use

[Actual result]:

- The profile appears as if it is in use by another app even though that is not the case

[Regression range]:

- This is not a regression. The issue occurs since the feature was implemented in Fx 67.0a1.

[Additional notes]:

- The issue does not occur on Ubuntu or macOS.
- No other windows app is using the deleted profile folder. There is no other Fx instance running to use the profile

Figure 4.6: BugZilla report 1546326 video before editing: 16s mark

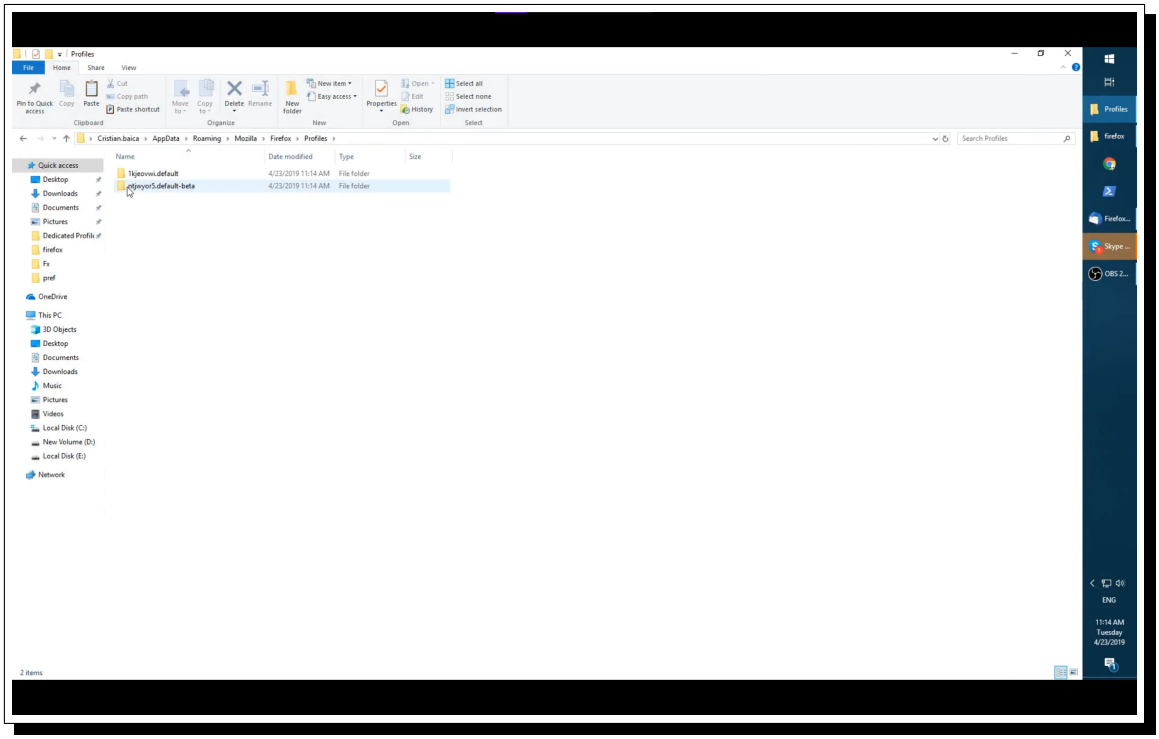


Figure 4.7: BugZilla report 1546326 video before editing: 48s mark

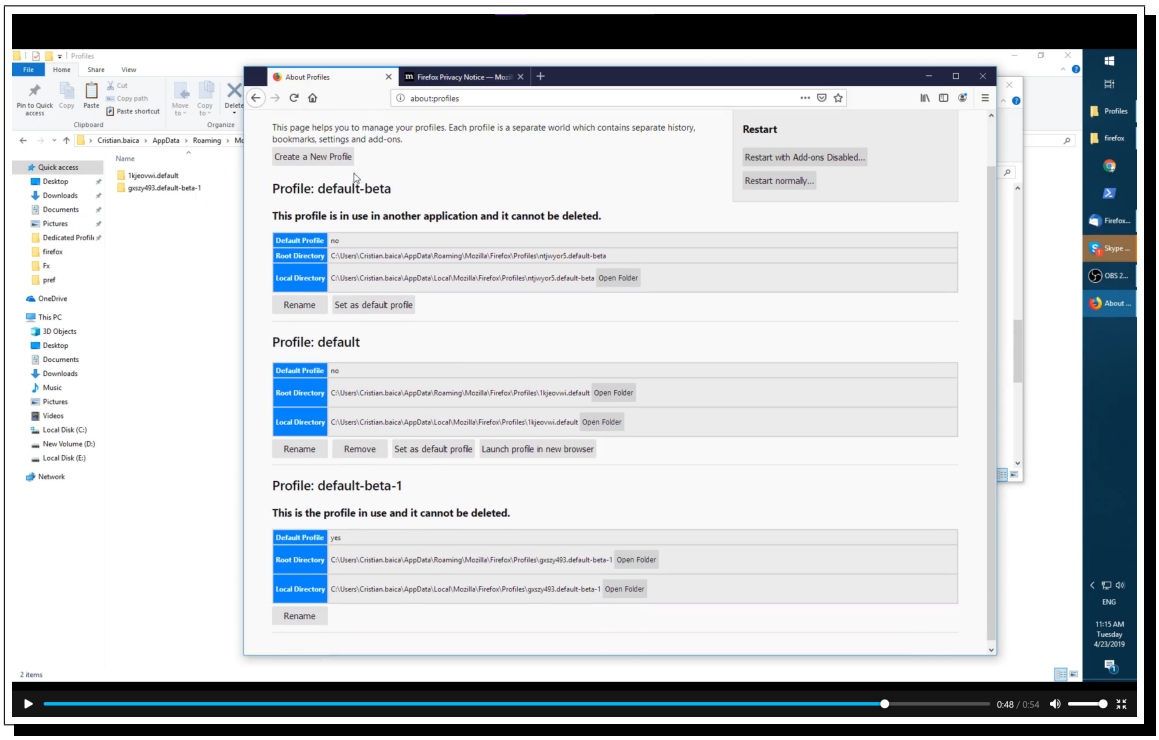


Figure 4.8: BugZilla report 1546326 video after editing: 16s mark

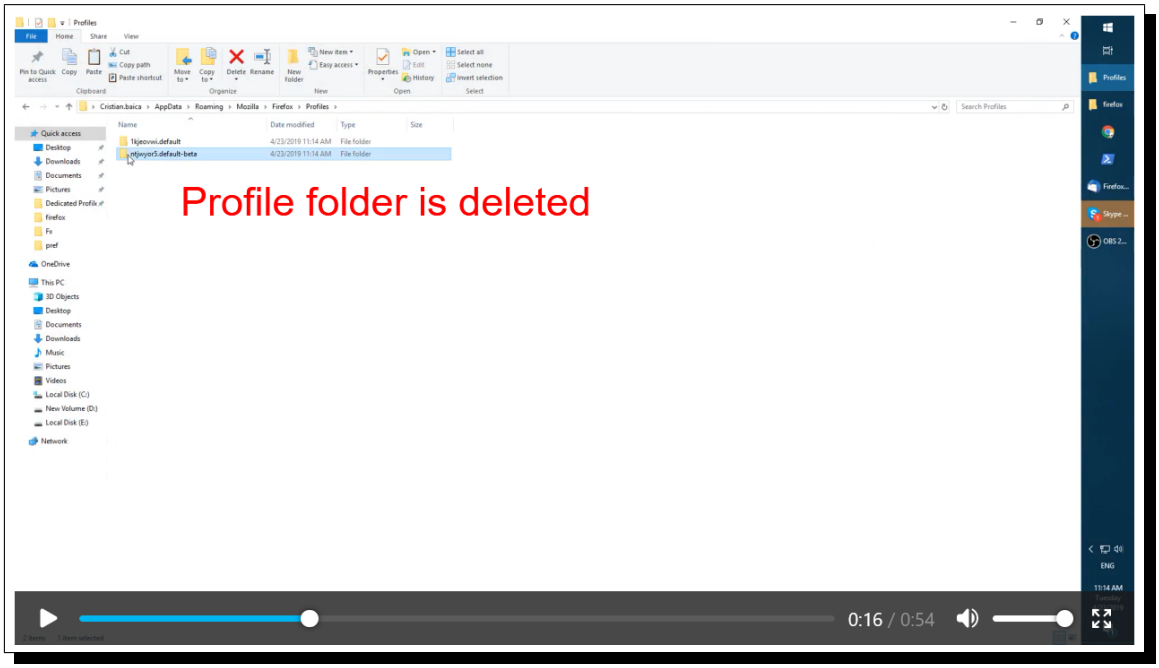
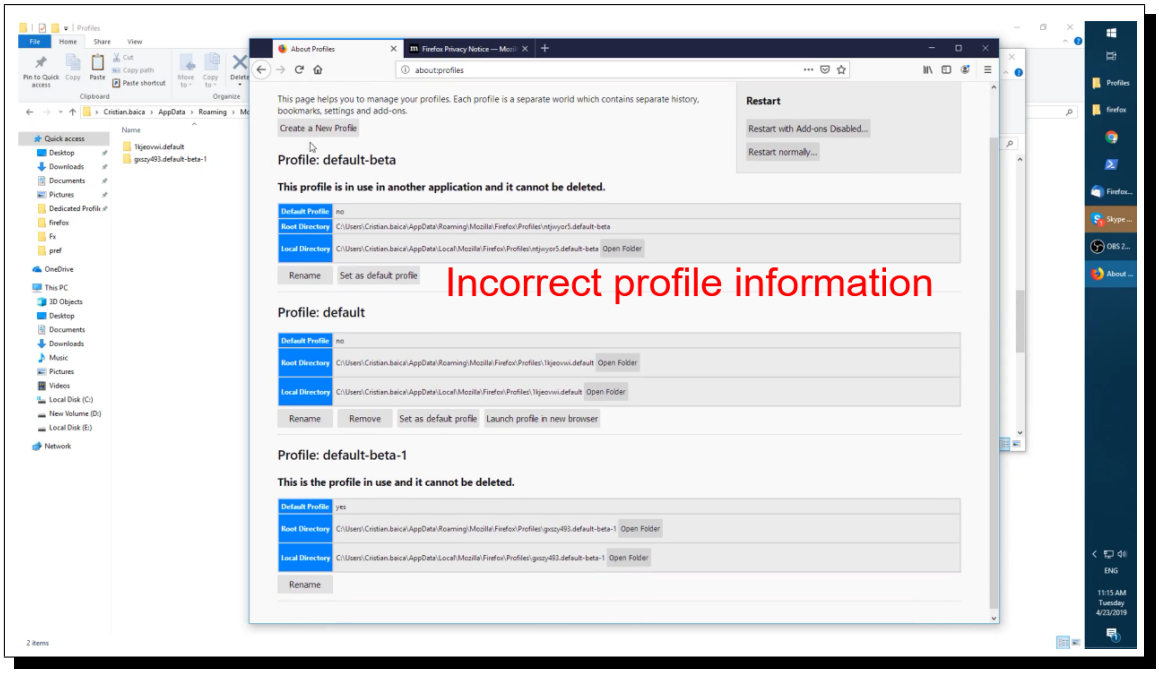


Figure 4.9: BugZilla report 1546326 video after editing: 48s mark



4.3 BugZilla Report 1028819

I selected BugZilla report 1028819, titled "[Candy Crush] Index page's audio issue", as a third example. This bug is an issue with Candy Crush, a mobile game that came pre-installed on devices running Firefox OS—the game plays audio for a second or two after closing the app when it instead should stop playing on app shutdown. The reporter included steps to reproduce, as well as expected and actual behavior, all of which are shown in Figure 4.10. I chose this bug to illustrate the extent of VideoLab's ability—adding annotations on a video with a primarily auditory focus provides a visual component that can further cue the developer on where exactly the behavior is occurring.

Because the nature of this bug is primarily auditory rather than visual, in this example I will show only a frame of the video attachment after it has been edited with VideoLab and explain the context of the edit. As shown in Figure 4.11, I have added a textual annotation on top of the video when the Candy Crush game has closed, but the audio from the game

Figure 4.10: BugZilla report 1028819 reproduction steps

```
STR:
1) Download Candy Crush, version 0.2.1.
2) Start the game


Observed:
the audio/music playback on index page has noise

Expected:
should playback normally.
```

Figure 4.11: BugZilla report 1028819 video after editing

continues to play for another second. This annotation provides an additional visual marker for a responding developer on where the defective behavior is occurring.

Chapter 5

Evaluation Plan

In this chapter, I propose a plan of evaluation to assess the ability of VideoLab to allow users to easily edit videos, as well as the ability of developers to ascertain information from video clips edited with VideoLab. The remainder of the chapter is organized as follows: Section 4.1 proposes questions to guide assessment, and Section 4.2 details the stages of the evaluation plan by explaining how data should be collected and analyzed.

5.1 Questions for Assessment

I propose several assessment questions for the evaluation of VideoLab. These questions serve to guide the evaluation of the tool and investigate whether the prototype could provide users with a smoother video editing experience in pursuit of supplementing the bug reporting and resolution process.

1. **How effective is VideoLab in allowing users to edit videos?**

   This question is intended to guide a design critique of VideoLab by presenting users

with a task and observing how they achieve the goal—in this case, to edit and export a pre-recorded video. I will evaluate the effectiveness of the tool on the results of the design critique, as well as the presence of any issues that may arise with the tool. Another metric on which to evaluate the effectiveness of VideoLab is to determine if videos produced by the tool can convey information more easily alongside a bug report than a report with no video attachment.

2. **Who finds VideoLab most effective for creating annotated videos?**

   As an extension of the previous question, I want to determine what demographic would find VideoLab most useful. Though one main objective is to maintain usability of the tool regardless of reporting experience level, the possibility of one group of reporters preferring the tool over another group may arise. Perhaps more experienced bug reporters would prefer not to use the tool while less experienced reporters find it useful, or vice versa.

3. **Is there a difference in the way inexperienced end-users versus experienced end-users use VideoLab?**

   This question serves to evaluate if VideoLab meets the previously stated objective of facilitating communication of defective behavior among end-users and developers, as users should not find difficulty in using the tool regardless of reporting experience.

4. **Is VideoLab effective in helping developers understand bug reports better, for all types of bug reports?**

   The design choices in developing VideoLab were primarily made with user-facing issues in mind—these tend to manifest in graphical user interfaces or other similar interface-type environments that can easily be shown and annotated in a video. However, I am interested in evaluating VideoLab for other, less visual types of bugs to see if VideoLab provides comparable ease of communication.

Together, these questions can act as a baseline for a thorough evaluation of VideoLab and work toward determining if the tool meets the goals I have set.

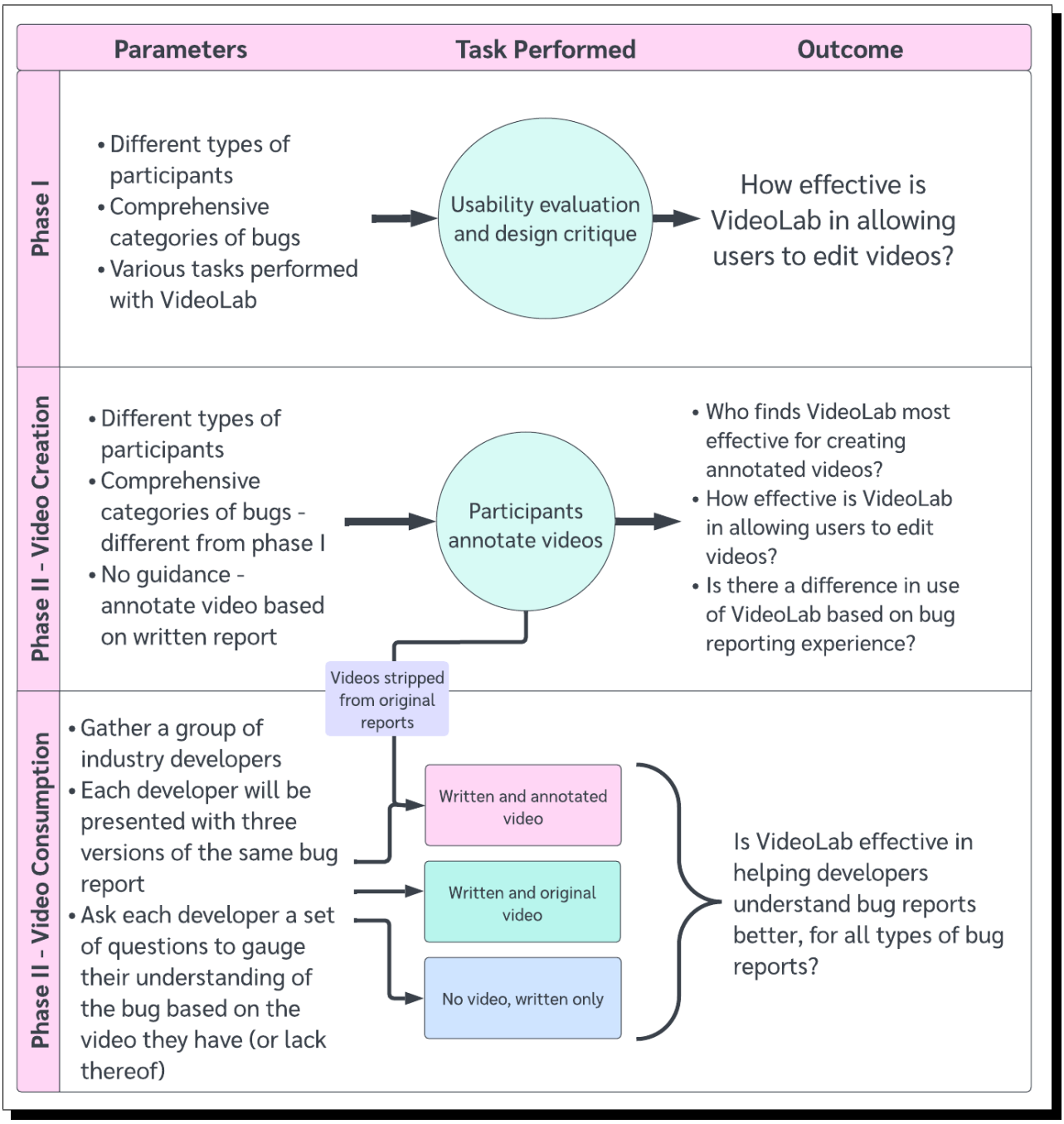
5.2 Evaluation Stages

To answer the assessment questions proposed above, I propose a two-stage evaluation process to examine the effectiveness of VideoLab. The stages of the evaluation consist of an initial design critique phase to evaluate the usability of VideoLab as an application, followed by a two-step second phase involving video creation by participants and consumption and evaluation by industry professionals of the resulting videos. Figure 5.1 shows the experimental design in its entirety, with the parameters, tasks performed, and outcome of each phase. Once this experiment is set in motion, it is likely to be updated and changed iteratively based on the experiences of the participants and how early results do and do not help in answering the questions proposed.

5.2.1 Phase I: Design Critique and Usability Evaluation

The first stage of VideoLab's evaluation is gathering initial feedback and observation data from a collection of participants of various levels of technical background and bug-reporting experience. The goal is to evaluate the usability of the tool and examine whether the tool meets the objective as stated in Chapter 3, particularly regarding ease of use of the video annotations. Using the design critique method [1], I will evaluate different groups of end-users to ensure a comprehensive feedback process, ensuring the tool's ease of use. Examples of different groups of end users include industry developers with several years of experience, users who contribute regularly to open-source projects, and users who have no bug reporting experience. This first step will allow an initial impression of the features of the tool, and what

Figure 5.1: Experiment Design



needs to be changed to better assist end-users in the process of creating video attachments.

In order to evaluate VideoLab's capability in its entirety, the participants will be asked to complete different types of tasks, such as placing one annotation on top of a video clip, placing several annotations on a video clip, deleting and re-uploading a video clip to be edited again, trimming a clip, and trimming a clip after placing an annotation on top.

As for the videos that would be included in this evaluation, I will compile a set of videos from bug reports from Mozilla's bug tracker, BugZilla. This set of videos will include videos of different lengths. Though finding video attachments for non-GUI issues might be more difficult due to the less visual nature of the bug, I will attempt to include as many different types of bugs as possible in this evaluation.

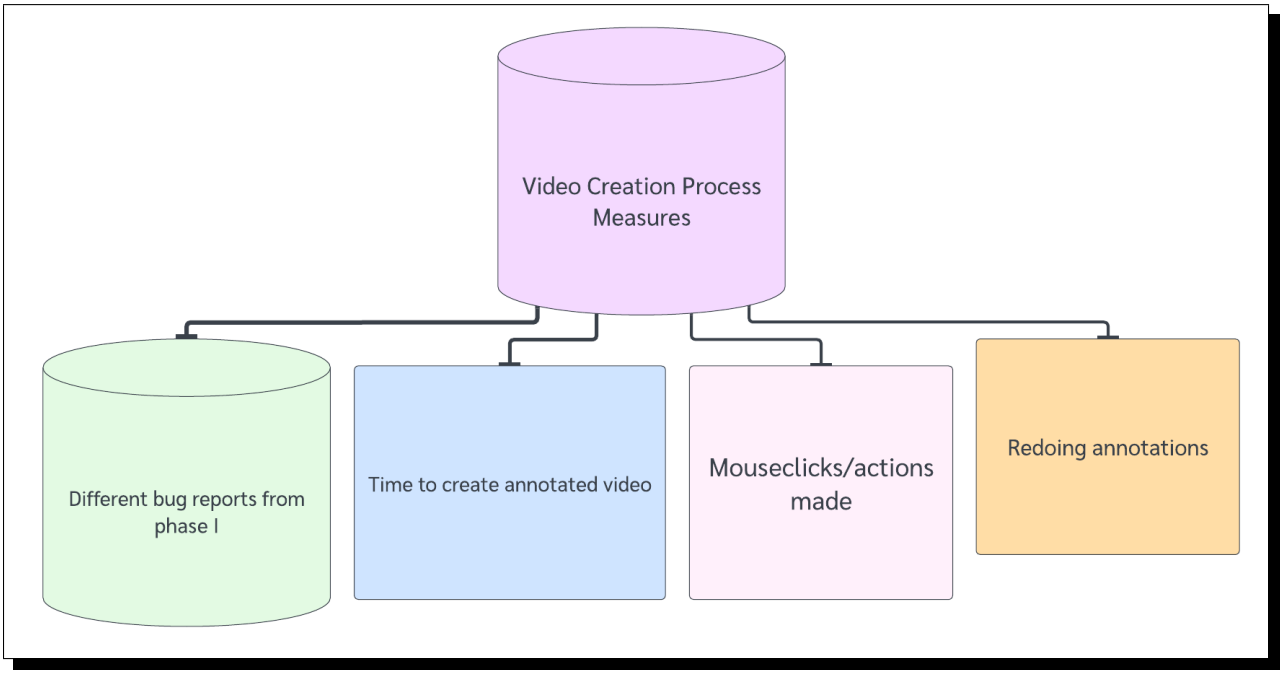
The results of the design critique and usability evaluation from this phase will allow me to answer the question of how effective VideoLab is in allowing end-users to edit videos.

5.2.2 Phase II, Video Creation

In the first portion of Phase II, I will gather a new, comprehensive set of bug reports for participants to annotate. As in Phase I, I plan to have different types of participants annotating videos, to represent varying types of end users. However, with this phase, I will not give participants any guidance on how to annotate the video—they will be asked to annotate the video based on what is written in the accompanying report.

While the participants annotate the videos, I will be keeping track of various experimental measures, shown in Figure 5.2. I will record how long participants take to annotate their videos, how many mouseclicks or other actions they took to achieve their goal, and whether or not they had to redo annotations. I will also ask participants to record their emotional state at regular intervals in order to get a sense of whether VideoLab provides an easy video

Figure 5.2: Video Creation Experimental Measures



editing experience.

While the focus of Phase II is mainly on the comparative experiment described in the next section, there is value in examining the creation process in detail. In particular, doing so can help answer the questions of who benefits most from using VideoLab, how effective VideoLab is in allowing users to edit videos, and whether there is a difference in use of VideoLab based on bug reporting experience.

5.2.3 Phase II, Video Consumption

The next stage of Phase II is to collect feedback from industry developers on the ability of a video edited by VideoLab to communicate information necessary to a bug report. I intend to gather the edited videos from the first part described in the previous section, along with the original bug reports that accompanied the videos, conduct a comparative experiment by

asking industry software developers to evaluate the effectiveness of the provided reports and complete a survey gauging their understanding.

Each developer will evaluate three versions of the bug reports: the original as it is written with the video attached, the written report only, and the original written report with the annotated video from the video creation section. In order to reduce the threat to validity of a learning effect, I will employ counterbalancing by showing the developers a different report for each category, as well as showing each category in a different order. I plan to ask the developers the following assessment questions for the evaluation of the video attachments:

1. **Does the video attachment make the bug report easier to understand?**

   This question aims to evaluate the effectiveness of VideoLab in easing end-users' ability to communicate information, similar to the question proposed for the first stage. The results of this question will be particularly useful in comparing the groups with the original video attachments versus the VideoLab-edited attachments. For the group with no video attachment access, this question would be rephrased to ask if the report itself is easy to understand.

2. **Is the report still missing useful information?**

   This question serves to further address the effectiveness of VideoLab in encouraging easier communication for end-users, particularly to evaluate if the video attachment communicates sufficient information that may be left out of the bug report. This question would be asked as-is to the group without access to video attachments.

3. **Does the video attachment prompt any questions about the report?**

   This question intends to address the issue of video attachments potentially increasing resolution time, particularly through increased exchange between reporter and developer. This increased exchange may be prompted by additional questions that may arise as a result of including the video in the bug report. For the group without video

attachments, they would be asked if the written report is missing any information.

In addition to gathering perceptions of the reports, I plan to conduct an experiment aimed to measure metrics related to the developers' understanding of the report, such as level of understanding of the report and ability to describe the report in as much detail as possible, all with the intent to set up resolution of the issue. Each developer will be given the same set of questions, with the appropriate modifications made to questions for the reports with no video access. The tasks I will give are as follows:

1. Rate your initial understanding of the defective behavior occurring in this report, 1 being "I don't understand this at all" and 10 being "I can come up with a solution for this report right now":
2. Follow the reproduction steps as they are written, before watching the video. Were you successful in reproducing the bug? Did you have trouble, or feel anything was missing?
3. After watching the video, try reproducing the behavior again. Were you successful this time? If so, what additional information did the video provide to you?
4. Rate your understanding of the defective behavior after watching the video and reading the report in more detail, with 1 being "I don't understand this at all" and 10 being "I can come up with a solution for this report right now":

These questions and associated tasks given to each developer specifically aim to gauge their understanding of how to approach the bug both before and after watching the edited video attachment. I will compare the responses from the experimental debrief, as well as feedback from the first phase, and aim to draw a conclusion on if VideoLab is effective for all types of bugs. The first evaluation stage observes this objective from the perspective of the end-users, while the second stage does so from the perspective of the developers.

Chapter 6

Useful Future Extensions

In Chapter 3, I outlined my design decisions and implementation details for VideoLab, which currently largely remains a prototype and proof of concept rather than a fully fleshed-out application due to development time constraints. In this chapter, I describe useful extensions for VideoLab that I had envisioned to add to its functionality so to bring the application closer to achieving the objective outlined in Chapter 3.

6.1 In-App Screen Recording

To maintain my objective of making bug reporting and video editing as straightforward yet informative of a process as possible, I wanted end-users to be able to conduct the entire video editing process within the app, without having to install or open any other applications to supplement the process. A useful extension of VideoLab's capability would be to include screen recording of defective behavior as a first step before the user annotates the video. While it is useful to be able to upload any video recording, by incorporating in-app video capture, the app can guide the user to what they should be recording.

The home screen of VideoLab prompts the user to select an application to record, and displays a red button that says "Start Recording." This functionality was not implemented in full, but the idea behind these elements is that the application would be able to list the applications currently running on the user's machine, similar to Windows Task Manager, and the user could select the application displaying defective behavior. VideoLab would then take the user to a new page to begin screen recording right inside VideoLab. I would augment the recording interface with instructions, such as:

- Don't record the entire interaction.
- Show steps leading up to the application failure.
- Capture the stack trace in your video recording.

Providing guidance to the user in this way as they record can in and of itself greatly improve the informational richness of the video being produced.

6.2 Voiceovers

Throughout this thesis, I have explained the importance of visual media as a method of communication and its strengths over solely using textual or verbal descriptions. However, a verbal description superimposed onto a visual medium may potentially heighten comprehension by the viewer [17], suggesting that the inclusion of a voiceover over a video attachment for a bug report could provide additional context and relevant information to a receiving developer over either a video or a verbal account on its own. Thus, a potentially beneficial future extension of VideoLab's capabilities would be to allow the end-user to record themselves describing what is happening in the video. By providing a verbal description of the

video's events in accompaniment with the visual information, the end-user can provide additional relevant detail and context to the defective behavior occurring in the video, especially if something in the video is not as readily discernible.

This functionality could be implemented by adding it into the editing palette. There are several ways users could include voiceovers—the tool could prompt the user to play the video and speak over it, or they could record their own voiceover snippets and drop them into the video at certain points, similar to users dragging and dropping annotations on top of the editing area.

6.3 Video Blurring

Because many video attachments to bug reports consist of screen recordings, either of the reporter's phone screen or desktop, there is the potential for sensitive or personally identifying information to appear in the recording. A useful feature to add to VideoLab's video editing tools would be the ability to blur out sensitive information. This could potentially be achieved by adding a solid rectangle to the annotations palette to allow users to cover information by rendering the rectangle on top, or by adding in a separate editing function that enables users to blur sections of the video that they do not want visible to the developer viewing their report.

6.4 Steps Recorder Integration

An interesting feature of the Windows operating system is the Steps Recorder tool [1], which

[1]https://support.microsoft.com/en-us/windows/record-steps-to-reproduce-a-problem-46582a9b-620f-2e36-00c9-04e25d784e47

records the steps a user takes to reproduce a problem they are experiencing. These steps include actions like mouseclicks, typing text, and various other actions that may be difficult to capture in a video recording. VideoLab could integrate with a tool like Steps Recorder to automatically create written annotations for mouse and keyboard actions, as well as prompting the user to attach their Steps Recorder log and generating written reproduction steps to be submitted alongside the video.

6.5 Selecting Annotation Duration

Presently, the functionality that allows users to place an annotation and determine how long it appears on the video prioritizes a simple design with little effort on the part of the user with regard to choosing and entering values. There are a number of ways this design could be altered and extended to further prioritize this goal.

One way is to eliminate the need for the user to enter a starting position value in the dialogue box that appears after placing the annotation. My initial plan was to allow the user to move within the clip to whatever point they want the annotation to begin appearing, and then select a duration. However, I found that attempting to implement that functionality was technically more challenging than I had anticipated, so I opted to allow the user to type it in instead.

Another way to alter this design in pursuit of maintaining a focus on simplicity is to remove the modal-style box and include the items in the box in some sort of tooltip or popup menu that appears alongside the annotation after it is placed. Similar to the starting timestamp design, using a tooltip was my original idea for this area, but proved to be technically difficult to implement. Additionally, using a tooltip did not supply me with enough space to fit the buttons comfortably, so I opted for the dialogue box design instead in an effort to maintain

readability over attempting to crowd the buttons into a small space.

Both of these implementations offer a more streamlined approach over the current implementation. However, these present some limitations. If the user misses their target starting point by a second or two, they have to delete the shape and try again, which necessitates greater effort.

6.6 Video Editing Timeline

Another option for extending the functionality of VideoLab in a more robust way, as well as improving upon selection of annotation duration, would be to implement a timeline-style interface. The timeline interface is standard in most modern video editing programs. Implementing it within VideoLab would condense the interface and make it easier to navigate—annotations could be dragged and dropped onto the timeline, and the duration of the annotation's appearance could be modified within the timeline rather than with the use of buttons. Additionally, the video could be dragged onto the timeline after uploading, and trimming the video would be directly implemented into the timeline similar to its function in traditional video editing software—a user could drag the ends of the video clip on the timeline to make it longer or shorter. This feature could provide a sizeable payoff in terms of a more concise interface and better usability, as most users are either familiar with the timeline interface already or could learn how to use it.

6.7 Bug Report Exporting and Submission

In order to provide a more complete bug reporting experience for users of VideoLab, another beneficial extension of its capabilities is to add the ability for end-users to provide additional

textual information not provided by the video, such as a bug summary or title, reproduction steps, a description of expected behavior of the application–since the video already displays the actual behavior–and perhaps some information that the application can automatically capture. Some examples of automatically captured information are the application's version, the operating system of the machine, or a crash dump generated by the defective application. The user would then be able to export both the video and the text of the bug report together and submit to their desired platform. This functionality could be implemented with the addition of another screen after the editing stage, allowing the user to enter all of the necessary information for a bug report into text fields and export it as a text, JSON, or CSV file alongside the video clip.

A more ambitious direction for this extension would be to allow an end-user to submit directly to the bug tracking system of choice from the tool, rather than collecting all of the information from the user but then to require them to submit it themselves. Though this functionality would be more intensive to implement, it would contribute to the goal of ease of communicating defective behavior.

Regardless of how this feature would be implemented, this capability would promote the ease of use of the tool and bug reporting in general, since the entire process can be completed inside a single, flexible tool, eliminating the need for an end-user to use several different applications or navigate to several different websites and do more work to submit a bug report.

Chapter 7

Conclusion

Managing bug reports is a costly and unfortunately necessary component of the software development process. In handling these reports, developers must manage user-submitted bug reports, which can prove to be time-consuming. The goal of VideoLab is to enable users to effectively communicate defective application behavior to developers through annotated video, meant to be used as supplementary information to bug reports. In designing and implementing a prototype of VideoLab, this thesis makes the following contributions: a pared-down video editor specifically intended to supplement bug reports, a set of dedicated annotations that focus on communicating information in videos for bug reports, and a detailed future assessment plan. In addition, VideoLab aims to bridge the communication gap between end-users and developers by promoting use of visual media to communicate defective behavior as a complement to typical textual or verbal descriptions. I additionally provided guidance for more features that VideoLab could greatly benefit from in order to achieve the stated objective.

Bibliography

[1] L. Alabood, Z. Aminolroaya, D. Yim, O. Addam, and F. Maurer. A systematic literature review of the design critique method. *Information and Software Technology*, 153:107081, 2023.

[2] P. Ardimento and C. Mele. Using Bert to predict bug-fixing time. In *2020 IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS)*, pages 1–7, 2020.

[3] M. Bakopoulos, S. Tsekeridou, E. Giannaka, Z.-H. Tan, and R. Prasad. Mobile video annotation for enhanced rich media communication during emergency handling. In *Proceedings of the 4th International Symposium on Applied Sciences in Biomedical and Communication Technologies*, ISABEL '11, New York, NY, USA, 2011. Association for Computing Machinery.

[4] C. Bellman, A. Seet, and O. Baysal. Studying developer build issues and debugger usage via timeline analysis in visual studio ide. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 106–109, 2018.

[5] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, page 308–318, New York, NY, USA, 2008. Association for Computing Machinery.

[6] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful ... really? *2008 IEEE International Conference on Software Maintenance*, pages 337–345, 2008.

[7] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 229, 2013.

[8] L. Cheng, E. Murphy-Hill, M. Canning, C. Jaspan, C. Green, A. Knight, N. Zhang, and E. Kammer. What improves developer productivity at google? code quality. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1302–1313, 2022.

[9] N. Cooper, C. Bernal-Cárdenas, O. Chaparro, K. Moran, and D. Poshyvanyk. It takes two to tango: Combining visual and textual information for detecting duplicate video-

based bug reports. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 957–969, 2021.

[10] M. Fazzini, K. Moran, C. Bernal-Cardenas, T. Wendland, A. Orso, and D. Poshyvanyk. Enhancing mobile app bug reporting via real-time understanding of reproduction steps. *IEEE Transactions on Software Engineering*, page 1, 2022. Publisher Copyright: IEEE.

[11] S. Feng and C. Chen. Gifdroid: Automated replay of visual bug reports for android apps. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1045–1057, New York, NY, USA, 2022. Association for Computing Machinery.

[12] D. R. Goldman. *A Framework for Video Annotation, Visualization, and Interaction.* PhD thesis, USA, 2007. AAI3275872.

[13] A. J. Ko and P. K. Chilana. How power users help and hinder open bug reporting. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, page 1665–1674, New York, NY, USA, 2010. Association for Computing Machinery.

[14] P. Krieter and A. Breiter. Analyzing mobile application usage: Generating log files from mobile screen recordings. In *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '18, New York, NY, USA, 2018. Association for Computing Machinery.

[15] D. Lin, C.-P. Bezemer, and A. E. Hassan. Identifying gameplay videos that exhibit bugs in computer games. *Empirical Software Engineering*, 12 2019.

[16] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk. Auto-completing bug reports for android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 673–686, New York, NY, USA, 2015. Association for Computing Machinery.

[17] N. M. Murray, L. A. Manrai, and A. K. Manrai. How super are video supers? a test of communication efficacy. *Journal of Public Policy & Marketing*, 17(1):24–34, 1998.

[18] M. Nayebi. Eye of the mind: Image processing for social coding. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 49–52, Los Alamitos, CA, USA, oct 2020. IEEE Computer Society.

[19] L. D. Panjer. Predicting eclipse bug lifetimes. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pages 29–29, 2007.

[20] M. S. Rakha, C.-P. Bezemer, and A. Hassan. Revisiting the performance of automated approaches for the retrieval of duplicate reports in issue tracking systems that perform just-in-time duplicate retrieval. *Empirical Software Engineering*, 23:2597–2621, 2018.

[21] L. Rello, M. Pielot, and M.-C. Marcos. Make it big! the effect of font size and line spacing on online readability. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, page 3637–3648, New York, NY, USA, 2016. Association for Computing Machinery.

[22] R. Schusteritsch, C. Y. Wei, and M. LaRosa. Towards the perfect infrastructure for usability testing on mobile devices. In *CHI '07 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '07, page 1839–1844, New York, NY, USA, 2007. Association for Computing Machinery.

[23] M. Soltani, F. Hermans, and T. Bäck. The significance of bug report elements. *Empirical Software Engineering*, 25(6):5255–5294, Nov 2020.

[24] Y. Song, J. Mahmud, Y. Zhou, O. Chaparro, K. Moran, A. Marcus, and D. Poshyvanyk. Toward interactive bug reporting for (android app) end-users. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 344–356, New York, NY, USA, 2022. Association for Computing Machinery.

[25] Y. Tian, C. Sun, and D. Lo. Improved duplicate bug report identification. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 385–390, 2012.

[26] A. N. Tuch, J. A. Bargas-Avila, K. Opwis, and F. H. Wilhelm. Visual complexity of websites: Effects on users' experience, physiology, performance, and memory. *International Journal of Human-Computer Studies*, 67(9):703–715, 2009.

[27] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan. An empirical study on factors impacting bug fixing time. In *2012 19th Working Conference on Reverse Engineering*, pages 225–234, 2012.

[28] J. M. Zhang, F. Li, D. Hao, M. Wang, H. Tang, L. Zhang, and M. Harman. A study of bug resolution characteristics in popular programming languages. *IEEE Transactions on Software Engineering*, 47(12):2684–2697, 2021.