Reverse Engineering Time-Series Interaction Data from Screen-Captured Videos

Lingfeng Bao[1], Jing Li[2], Zhenchang Xing[2], Xinyu Wang [§1], and Bo Zhou[1]

[1]College of Computer Science, Zhejiang University, Hangzhou, China
[2]School of Computer Engineering, Nanyang Technological University, Singapore
{lingfengbao, wangxinyu, bzhou}@zju.edu.cn; {jli030, zcxing}@ntu.edu.sg;

***Abstract*—In recent years the amount of research on human aspects of software engineering has increased. Many studies use screen-capture software (e.g., Snagit) to record developers' behavior as they work on software development tasks. The recorded task videos capture direct information about which activities the developers carry out with which content and in which applications during the task. Such behavioral data can help researchers and practitioners understand and improve software engineering practices from human perspective. However, extracting time-series interaction data (software usage and application content) from screen-captured videos requires manual transcribing and coding of videos, which is tedious and error-prone. In this paper we present a computer-vision based video scraping technique to automatically reverse-engineer time-series interaction data from screen-captured videos. We report the usefulness, effectiveness and runtime performance of our video scraping technique using a case study of the 29 hours task videos of 20 developers in the two development tasks.**

I. Introduction

It has long been recognized that the humans involved in software engineering, including the developers as well as other stakeholders, are a key factor in determining project outcomes and success. A number of workshops and conferences (e.g. CHASE, VL/HCC) have focused on human and social aspects in software engineering. An important area of these studies is to investigate the capabilities of the developers [1–3], their information needs in developing and maintaining software [4–6], how developers collaborate [7, 8], and what we can do to improve their performance [9–12].

Different from software engineering research with technology focus, research that focuses on human aspects in software engineering adopts behavioral research methods widely used in humanities and social sciences [13]. The commonly used data collection methods in such human studies include questionnaire, interview, and observation. Among these data collection methods, observation can provide direct information about behavior of individuals and groups in a natural working context. It also provides opportunities for identifying unanticipated outcomes.

Two kinds of techniques have been commonly used to automatically record observational data in the studies of developer behavior: software instrumentation and screen capture. We can instrument software tools that the developers use to log the developers' interaction with the tools and the application content. For example, Eclipse IDE can record which refactorings the developers apply to which part of the code[14]. Instrumenting many of today's software systems is considerably complex. It often requires sophisticated reflection APIs (e.g., Accessability API or UI Automation API) provided by applications, operating systems and GUI toolkits [15, 16]. Furthermore, developers use various tools (e.g., IDE, web browsers) in software development tasks. Instrumenting all of these tools requires significant efforts.

Screen-capture techniques offer a generic and easy-to-deploy alternative to instrumentation. Screen-capture software can easily capture the developers' interaction with several software tools. We surveyed 25 papers that were published in top-tier software engineering conferences from 1992 to 2014. These papers have studied various human aspects in software engineering. Screen-capture techniques were commonly used to record the developers' behavior in these studies. However, many studies used video data mainly as qualitative evidence of study findings. Some studies [3, 4, 9] performed quantitative analysis of developers' behavior by manually transcribing and coding screen-captured videos. These studies provided deeper insight into the developers' behavior in various software development tasks. Such quantitative analysis was expensive and time consuming. It was reported that the ratio of video recording time and video analysis time was about 1:4-7.

As the amount of research on human aspects of software engineering has increased, there has been a greater need to come up with a solution to automatically extract software usage and application content data from screen-captured videos, in order to facilitate quantitative analysis of the developers' behavior in software development tasks. In this paper, we present a computer-vision-based video scraping technique to recognize window-based applications in screen-captured videos, and to extract application-specific content from the recognized application windows (e.g., code fragments in code editor, error messages in console output, URLs in address bar of web browser, and keywords in search box of search results page).

We conducted a case study to evaluate the usefulness, effectiveness, and runtime performance of our video scraping tool. Our study demonstrated the effectiveness of video scraping technique in reverse-engineering time-series interaction data (software usage and application content) from screen-captured

[§]Xinyu Wang is the corresponding author

978-1-4799-8469-5/15/$31.00 © 2015 IEEE

SANER 2015, Montréal, Canada

videos. Based on the extracted time-series interaction data, we conducted a quantitative analysis of the 20 developers' online search behavior in the two development tasks. Our study also identified the improvement space of the tool's runtime performance.

The remainder of the paper is structured as follows. Section II summarizes our survey of the use of scree-captured videos in the 25 studies on human aspects of software engineering. Section III describes the metamodel of application windows that our technique assumes. Section IV discusses technical details of our video scrapping technique. Section V reports our evaluation of the tool. Section VI reviews related work. Section VII concludes the paper and discusses the future work.

II. Use of Screen-Capture Videos in SE Studies

We performed a literature review using keywords such as "software engineering", "exploratory study", "empirical study" and/or "screen capture". From the search results, we surveyed 25 papers that studied human aspects of software engineering. Among these 25 papers, 13 papers studied and modeled the developers' behavior in various software development tasks, such as debugging [1, 3, 17], feature location [5], program comprehension [2, 4, 6, 11, 18–20], feature enhancement [17], and using unfamiliar APIs [12, 21]; 3 papers elicited information needs and requirements for improving software development tools [9, 22, 23]; 5 papers studied software engineering practices such as novice programming [10], pair programming [7], distributed programming [8], testing of plugin systems [24], and game development [25]; and 4 papers investigated visualization techniques of software data such as code structure [26, 27], program execution [28], and social relationship in software development [29].

Our survey showed that screen-capture techniques have been widely used to collect observational data in studying human aspects of software engineering, especially for modeling the developers' behavior in software development tasks and eliciting design requirements for innovative software development tools. Some studies [1, 3, 7, 20, 26, 27, 29] used think-aloud technique [30] to collect the data about the developers' behavior in the tasks. Think-aloud technique is obtrusive. It may affect the developers' normal behavior. A few studies [1, 7, 28] used human observers to observe and take notes of the developers' behavior. This human-observer approach does not scale well and may suffer from experimenter expectancy effect [13]. Studying software engineering practices (e.g., peer programming [7], game development [25], and plugin testing [24]) usually used survey and interview methods that can collect only self-reported qualitative data.

Although screen-capture techniques provide scalable and unobtrusive techniques to collect the developers' behavior data, the collected video data have been underused in many studies. A key reason for this underuse is the significant time and efforts required for manually transcribing or coding unstructured video data into software usage and application content data for the study purpose. 15 studies reported manual analysis of screen-captured videos in order to identify types of information the developers explored [3, 18, 19], information foraging actions [5, 9, 10, 17, 21–23], and patterns of developers' information behavior [4, 6, 11, 12, 20]. 4 papers [6, 9, 10, 23] of these studies reported the efforts required for manual coding of the collected screen-captured video data. The reported ratio of video recording time and analysis time was between 1:4-7, depending on the details and granularity of the information to be collected.

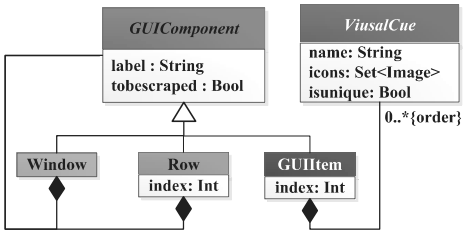


Fig. 1. The Metamodel of Application Windows

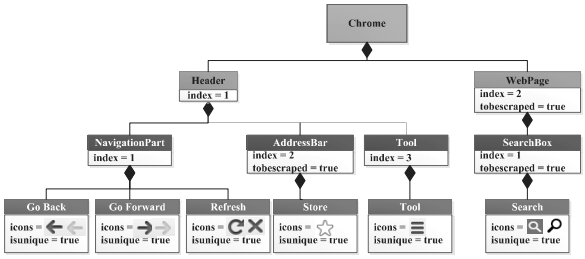
The most costly studies were to study fine-grained behavioral patterns in software development tasks (e.g., [5, 9]) because they required iterative open coding of screen-captured videos. For example, Ko and Myers [9] reported "analysis of video data by repeated rewinding and fast-forwarding". However, compared with qualitative data collection and analysis methods, such fine-grained studies of the developers' behavior can provide deeper insights into the outstanding difficulties in software development, and thus inspire innovative tool support to address these difficulties [31, 32].

Summary: Previous studies demonstrated the usefulness of screen-capture videos in studying human aspects of software engineering. However, to fully exploit the potentials of screen-captured video data in software engineering studies, there is a great need for automatic tool support that can reverse-engineer time-series interaction data (software usage and application content) from screen-captured videos.

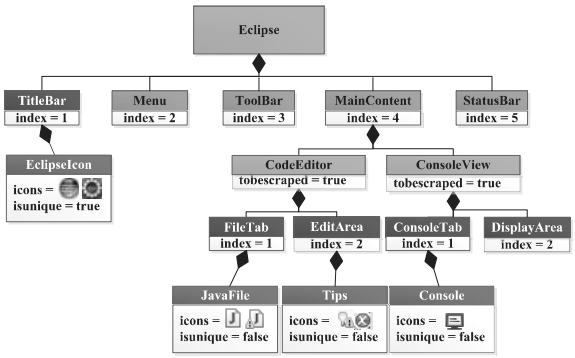
III. Definition of Application Window

A person recognizes an application window based on his knowledge of the window layout and the distinct visual cues (e.g., icons) that appear in the window. Our video scraping technique (*scvRipper*) requires as input the definition of application windows to be recognized in a screen-captured video. The definition of an application window "informs" the *scvRipper* tool with the window layout, the sample images of distinct visual cues of the window's GUI components, and the GUI components to be scraped once they are recognized.

Fig. 1 shows the metamodel of application windows. *scvRipper* assumes that an application window is composed of a hierarchy of GUIComponents. Rows and windows define the layout of the application window. A row or window can contain nested rows, nested windows, and/or leaf GUIItems. Rows and GUIItems have relative positions in the application window (denoted as $index$), while windows do not have. A GUIItem contains an order set of VisualCues. A VisualCue contains a set of sample images of the visual cue. If the application window can have only one instance of a VisualCue,



(a) Definition of Google Chrome Window



(b) Definition of Eclipse IDE Window

Fig. 2. Two Instances of Application-Window Metamodel

the $isunique$ of the VisualCue is $true$. The GUIComponents whose $tobescraped = true$ will be scraped from the application window in the screen-captured video.

Fig. 2 shows the definition of the Eclipse IDE and the Google Chrome window. The definition of the Eclipse window assumes that the Eclipse window consists of a GUIItem (TitleBar) and four rows (Menu, ToolBar, MainContent, and StatusBar) from top down. We omit the definition details of Menu, ToolBar and StatusBar due to space limitation. The TitleBar contains a unique VisualCue (Eclipse application icon). MainContent row may contain CodeEditor windows and ConsoleView windows. CodeEditor window contains FileTab and EditArea GUIItems. These two GUIItems contain non-unique visual cues (such as Java file icons, compile error icons). This definition instructs *scvRipper* to scrape CodeEditor and ConsoleView windows from the Eclipse window.

The definition of the Chrome window assumes that the Chrome window consists of two rows from top down: Header and WebPage. The Header contains three GUIItems from left to right: NavigationPart, AddressBar, and Tool. NavigationPart contains three VisualCues from left to right: GoBack, GoForward, and Refresh buttons. These buttons are unique in the Chrome window. The WegPage may contain a SearchBox GUIItem as commonly seen in search engine wegpages. A SearchBox has a unique Search button VisualCue. This definition instructs *scvRipper* to scrape AddressBar, SearchBox and WebPage from the Chrome window.

We have developed a configuration tool to aid the definition of application windows. The tool can define the hierarchy of GUIComponents, configure the attributes of GUIComponents, and attach sample images of visual cues to GUIComponents. Collecting sample images of visual cues may require certain efforts. However, this task usually needs to be done only once. The definition of an application window can be applied to screen-captured videos taken in different screen resolutions and window color schema, as neither window definition nor computer-vision techniques that *scvRipper* uses are sensitive to screen resolutions and window color schema.

IV. Video Scraping Technique

In this section, we first give an overview of our video scraping technique. We then detail the key steps.

A. Technique Overview

Fig. 3 presents the process of our video scraping technique. We have implemented our technique in a tool (called *scvRipper*) using $OpenCV$ (an open-source computer vision library). Our *scvRipper* tool takes as input a screen-captured video, i.e., a time-series stream of screenshots taken by screen-capture tools such as Snagit [1]. It produces as output a time-series interaction data (i.e., software usage and application content) reverse-engineered from the video. Our *scvRipper* tool essentially uses computer-vision techniques to transcribe a stream of screenshots that only human can interpret into a stream of interaction data that a computer can automatically analyze or mine for behavioral patterns.

First, *scvRipper* uses an image differencing technique [33] to detect screenshots with distinct content in the screen-captured video. This step reduces the number of screenshots to be further analyzed using computationally expensive computer-vision techniques. Next, the core algorithm of *scvRipper* processes one distinct-content screenshots at a time to recognize application windows in the screenshot based on the definition of application windows provided by the user. The recognized application windows identify software used at a specific time in the video. Then, *scvRipper* scrapes the GUIComponent images from the recognized application windows in the screenshot as specified in the definition of application windows. It uses Optical-Char-Recognition (OCR) technique to convert the scraped GUIComponent images into textual application content processed at a specific time in the video.

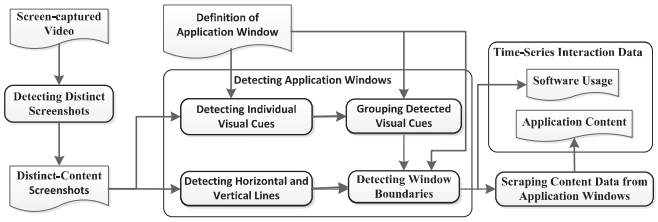


Fig. 3. The Process of Our Video Scraping Technique

The upper part of Fig. 4 shows an illustrative example of a screen-captured video. In this example, four distinct-content screenshots are identified at five time periods. The lower part of Fig. 4 shows the time-series interaction data

[1] http://www.techsmith.com/snagit.html

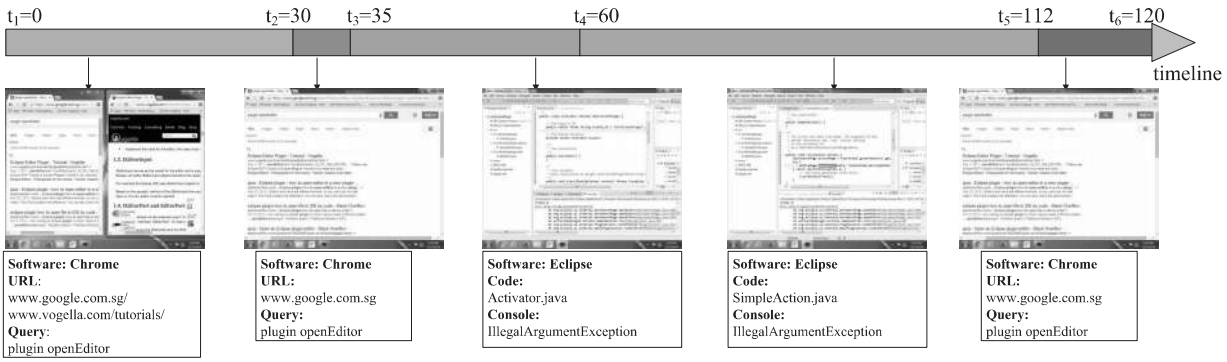


Fig. 4. An Illustrative Example Of a Screen-Captured Video and Video Scraping Results

reverse-engineered from these four distinct-content screenshots according to the definition of Eclipse IDE and Google Chrome window in Fig. 2. Bulky contents (e.g., web page, code fragment) are omitted due to space limitation. This time-series interaction data identifies the software tools that the developer used at different time periods. It also identifies the application content that the developer processed (such as search queries, visited websites, code fragments, and runtime exceptions) at different time periods.

B. Detecting Distinct-Content Screenshots

The screen capture tools can record a large number of screenshots (e.g., 30 screenshots per second). A sequence of consecutive screenshots may be the same, for example a person does not interact with the computer for a while. Or they may differ little, for example due to mouse movement, button click, or small scrolling. Thus, there is no need to analyze each screenshot in the screen-captured video.

To that end, *scvRipper* uses an image differencing algorithm [33] to filter out subsequent screenshots with no or minor differences in the screen-captured video. This produces a sequence of distinct consecutive screenshots, $s_1, s_2, ..., s_n$ where any two consecutive screenshots s_i and s_{i+1} are different, i.e., over a user-specified threshold (t_{diff}). The two non-consecutive screenshots can still be the same in this sequence of distinct consecutive screenshots. *scvRipper* uses image differencing technique again to identify distinct-content screenshots. *scvRipper* stores the traceability between a distinct-content screenshot and all the screenshots it represents during this image differencing process.

Take the screen-captured video in Fig. 4 as an example. The developer views two web pages side-by-side in the two Chrome windows. He then maximizes one of the Chrome windows. After a while, he switches from the Chrome window to an Eclipse IDE window. He opens two different methods in Eclipse and read the code. Next he switches from the Eclipse window back to the Chrome window. Assume this sequence of human-computer interaction takes 120 seconds. A screen-capture tool can record 600 screenshots at the sample rate 5 screenshots per second.

Given this stream of 600 screenshots, *scvRipper* can identify a sequence of five distinct consecutive screenshots as shown in Fig. 4. It can then identify that the screenshots at time periods $t_2 - t_3$ and $t_5 - t_6$ are the same. The screenshots at time periods $t_1 - t_2$ and $t_2 - t_3$ are similar but still different enough to be considered as two distinct-content screenshots. As such, *scvRipper* only needs to further analyze four distinct-content screenshots out of 600 raw screenshots.

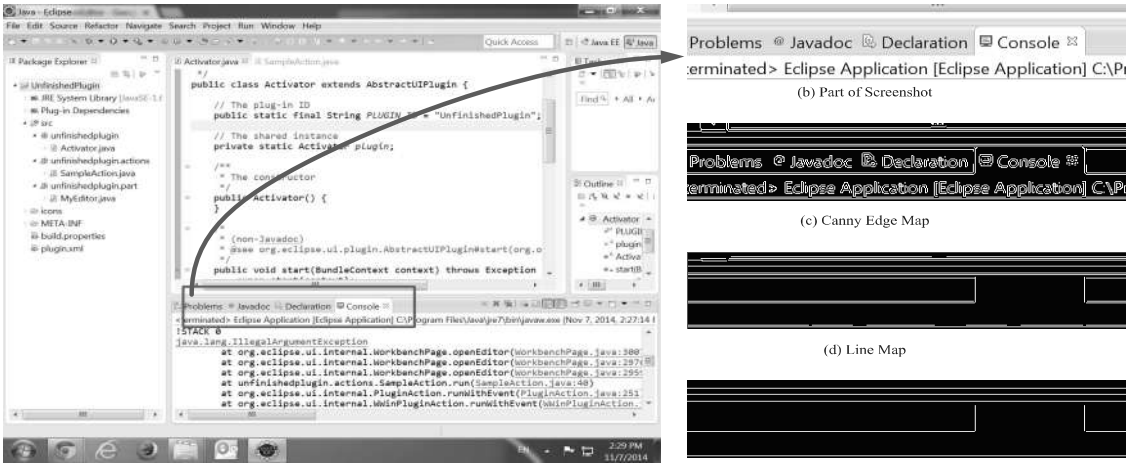
C. Detecting Application Windows

The core algorithm of *scvRipper* takes as input a distinct-content screenshot and the definition of application windows to be recognized in the screenshot. It recognizes application windows in the screenshot in four steps: 1) detect horizontal and vertical lines, 2) detect individual visual cues, 3) group detected visual cues, and 4) detect window boundaries. *scvRipper* can accurately recognize stacked or side-by-side windows.

1) Detecting Horizontal and Vertical Lines: Fig. 5 illustrates the process of detecting horizontal and vertical lines. Fig. 5(a) is the screenshot of the Eclipse window at time period $t_3 - t_4$ in Fig. 4. *scvRipper* assumes that an application window (or subwiondow) has explicit window boundaries and occupies a rectangular region in the screenshot. Thus, *scvRipper* first uses the canny edge detector [34] to extract the edge map of a screenshot. An edge map is a binary image where each pixel is marked as either an edge pixel or a non-edge pixel. Fig. 5(b) shows the canny edge map of the part of the screenshot in Fig. 5(a).

Then *scvRipper* performs two morphological operations (erosion and dilation) on the canny edge map. Erosion with a kernel (a small 2D array, also referred to filter or mask) [35] shrinks foreground objects by stripping away a small layer of pixels from the inner and outer boundaries of foreground objects. It increases the holes enclosed by a single object and the gaps between different objects, and eliminates small details. Dilation has the opposite effect of erosion. It adds a small layer of pixels to the inner and outer boundaries of foreground objects. It decreases the holes enclosed by a single object and the gaps between different objects, and fills in small intrusions into boundaries.

For horizontal lines, erosion followed by dilation with the kernel $[1]_{1\times K}$ (i.e., a horizontal line of K pixels) on the edge map remove the horizontal lines whose length is less than K. For vertical lines, erosion followed by dilation with the kernel is $[1]_{K\times 1}$ (i.e., a vertical line of K pixels) on the edge map remove the vertical lines whose length is less than K. These



(a) A Screenshot of Eclipse IDE Window (b) Part of Screenshot (c) Canny Edge Map (d) Line Map (e) Clustered Line Map

Fig. 5. An Example of Detecting Horizontal and Vertical Lines

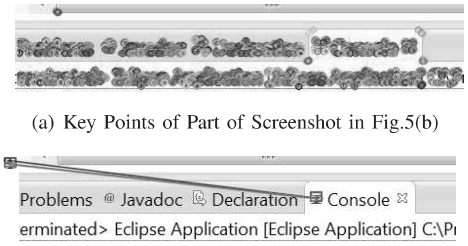
erosion and dilation operations generates a line map of the screenshot (see Figure 5(c)).

The horizontal (or vertical) lines in the line map can be very close to each other. Such close-by horizontal (or vertical) lines introduce noises and increase complexity to detect the window boundaries. Given a line map of the screenshot, *scvRipper* uses density-based clustering algorithm (DBSCAN [36]) to cluster the close-by horizontal (or vertical) lines based on their geometric distance and overlap. For each cluster of horizontal (or vertical) lines, *scvRipper* generates a representative line by choosing the longest line in the cluster and extending this line to the smallest start pixel position and the largest end pixel position of all the lines in the cluster.

2) Detecting Individual Visual Cues: scvRipper uses the samples of visual cues provided in the definition of an application window as image templates. It detects the distinct visual cues of an application in the screenshot using key point based template matching [37, 38]. Key point based template matching is an efficient and scale invariant template matching method. A key point in an image is a point where the local image features can differentiate one key point from another.

scvRipper uses the Features from Accelerated Segment Test (FAST) algorithm [39] to detect the key points of an image. It extracts the Speeded Up Robust Features (SURF) [38] of the detected key points. *scvRipper* detects the occurrences of a template image in a given screenshot by comparing the similarities between the key points of the template image and the key points of the screenshot [40]. Fig. 6(a) visualizes the key points image of the part of the screenshot in Fig. 5(b). The left corner of Fig. 6(b) visualizes the key points image of the visual cue of ConsoleView of Eclipse window. *scvRipper* detects the occurrence of this visual cue in the screenshot as indicated by the lines in Fig. 6(b)

The visual cues of an application are usually small icons. Some small icons may not always have enough key points, for example, the Java file icon of CodeEditor of Eclipse window. In such cases, *scvRipper* detects the visual cues in



(a) Key Points of Part of Screenshot in Fig.5(b)

(b) Key Point Matching

Fig. 6. An Example of Detecting Individual Visual Cues

a screenshot using template matching with alpha mask. The alpha mask of an image is a binary image used to reduces the effect of transparent pixels on the template matching. Given a visual cue image, its alpha mask, and the screenshot, *scvRipper* computes the normalized cross-correlation between the visual cue image and the subimages of the screenshot with the same size as the visual cue image [41]. The higher the normalized cross-correlation value, the more similar between the visual cue image and the subimages. *scvRipper* considers it as a match if the normalized cross-correlation value between the visual cue image and the subimage is greater than a user-specified threshold (usually a high threshold like 0.99).

3) Grouping Detected Visual Cues: A screenshot may or may not contain the application windows of interest. To determine if the screenshot contains the window(s) of a given application, *scvRipper* counts the number of the detected visual cues that belong to the application according to the definition of the application window. Multiple instances of the same type of VisualCues are counted once. If the number of the detected visual cues that belong to the application is more than $t_{app}\%$ (a user-specified threshold) of the number of VisualCues defined in the definition of application window, *scvRipper* considers that the screenshot contains the window(s) of the given application.

If the screenshot contains the application window(s) of interest, *scvRipper* uses normalized min-max cut algorithm [42] to group the detected visual cues into different application

windows, as the screenshot may contain two or more windows of the same application. Normalized min-max cut algorithm is an image segmentation technique that groups pixels into segments based on an affinity matrix of pairwise pixel affinities such as pixel color similarity and geometric distance. In our application of normalized min-max cut algorithm we define the affinity of the two detected visual cues as the possibility of the two visual cues belonging to the same application window.

If the two visual cues belong to two different applications (e.g., Eclipse versus Chrome) according to the definition of application windows, *scvRipper* sets their affinity at 0. If the two visual cues belong to the same application, *scvRipper* computes the affinity of the two visual cues based on the uniqueness of the visual cues, their relative positions, and their geometric distance.

If the two visual cues are the same type of VisualCue of an application and the $isunique$ of this type of VisualCue is $true$, *scvRipper* sets their affinity at 0. That is, it is impossible that these two visual cues belong to the same application window because the application window can have only one instance of this type of VisualCue. Fig. 7 shows the screenshot of the two side-by-side Chrome windows at time period $t_1 - t_2$ in Fig. 4. In this example, the affinity between the two detected "Go Back" visual cues (V_1 and V_3) is 0 because a Chrome window can have only one "Go Back" button. The same for the "Tool" visual cues(V_2 and V_4).

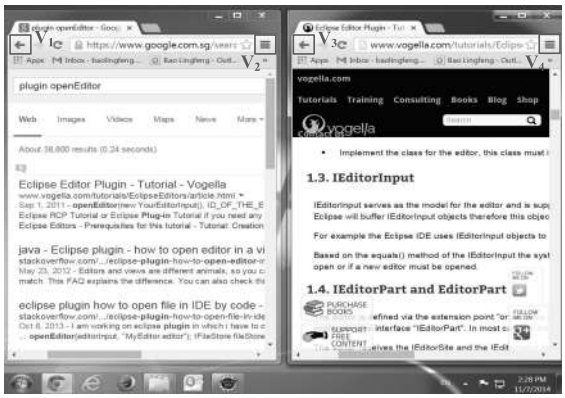


Fig. 7. An Example of Affinity Calculation

If the two visual cues are different types of VisualCues of an application, *scvRipper* compares the relative position of the two visual cues against the position constraints defined in the definition of the application window. If the relative position of the two visual cues is inconsistent with the position constraints, *scvRipper* set their affinity at 0. For example, the "Go Back" button is supposed to be at the left of the "Tool" button in a Chrome window. Thus, it is impossible that the detected "Go Back" button V_3 and the "Tool" button V_2 belong to the same Chrome window, because V_3 is at the right of V_2.

Given the two visual cues whose affinity is not yet set at 0 based on the uniqueness and relative positions of the visual cues, *scvRipper* computes their affinity as $e^{-(d_{ij}^2/\delta^2)}$ where d_{ij} is the distance between the center of the two visual cues V_i and V_j and δ is a term proportional to the image size. Intuitively,

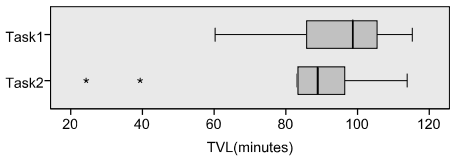


Fig. 9. The Statistics of Task Video Length (TVL)

the more distance between the two visual cues, the less likely the two visual cues belong to the same application window. In Fig. 7 the visual cues V_1 and V_3 (or V_2 and V_4) more likely belong to the same Chrome window than V_1 and V_4.

4) Detecting Window Boundaries: Given a group of detected visual cues belonging to an application window, *scvRipper* first calculates the smallest rectangle enclosing the group of detected visual cues. It then expands this smallest rectangle to find the bounding horizontal and vertical lines that form the bounding box of the group of detected visual cues. This bounding box is considered as the boundary of the application window. *scvRipper* records software usage at a specific time t in the screen-captured video in terms of the application window(s) present in the screenshot at time t. Once the boundary of an application window is determined, *scvRipper* further determines the boundary of the GUI components to be scraped within the application window boundary using the same method, based on the group of detected visual cues belonging to the to-be-scraped GUI components.

Fig. 8 shows the detected boundaries of the Eclipse window (at time period $t_3 - t_4$ in Fig. 4) and the Chrome window (at time periods $t_2 - t_3$ and $t_5 - t_6$ in Fig. 4). It also shows the detected boundaries of the to-be-scraped GUIComponents in the two windows. The detected boundaries are highlighted in the same color as that of the corresponding type of GUIComponent in Fig. 1.

D. Scraping Content Data from Application Windows

Based on the detected boundary of the to-be-scraped GUIComponents, *scvRipper* crops the portion of the screenshot and uses Optical-Character-Recognition (OCR) techniques (e.g., ABBYY FineReader) to convert image content into textual data. Fig. 8 presents an example of the image scraping results of the Eclipse window and the Chrome window. The OCRed textual data records what contents the developer works on at a specific time in the screen-captured video. For example, the scraped code snippet and the exception message show that the developer is editing the *Activator* class and he encounters the exception *IllegalArgumentException*. The scraped URL and search query show that the developer uses the Google search engine (domain name "google.com" in the URL) and his search query is "plugin openEditor".

V. Evaluation

We now report our evaluation of the usefulness, effectiveness and runtime performance of our *scvRipper* tool.

A. Data Set

Our evaluation was conducted on the screen-captured videos that we collected in our previous study of the developers'

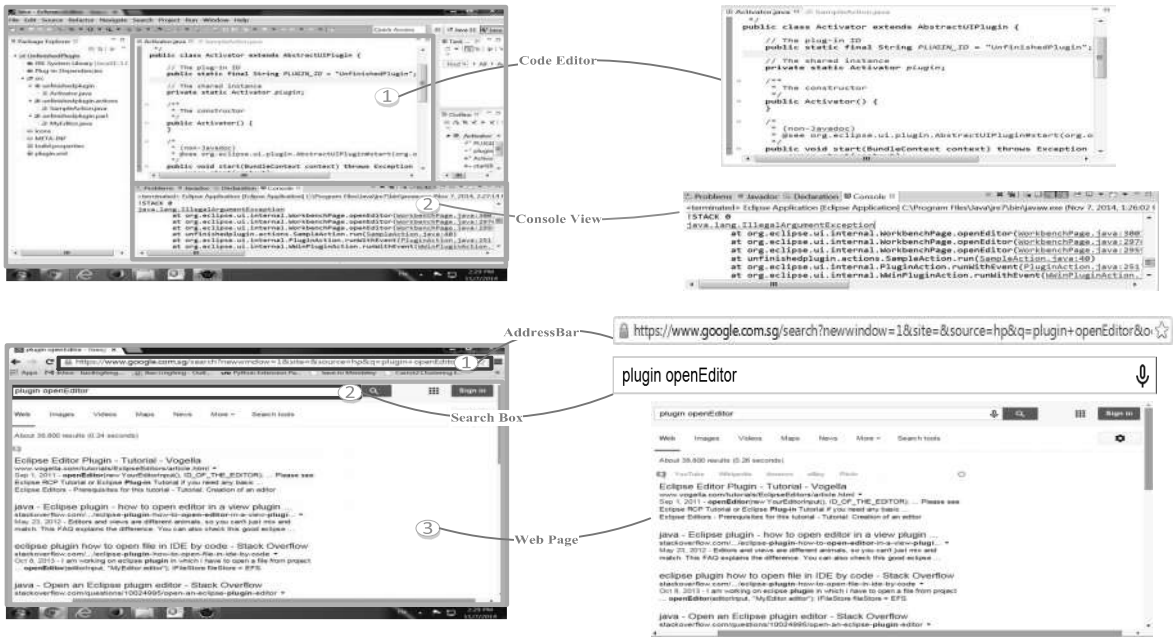


Fig. 8. An Example of Boundary Detection and Image Scraping Results

online search behavior during software development [4]. This previous study included two software development tasks: 1) develop a new P2P chat software, and 2) fix bugs and extends an existing Eclipse editor plugin. 11 graduate students were recruited in the first task, and 13 different graduate students were recruited in the second task from the School of Computer Science, Fudan University.

The participants were instructed to use a screen-capture software to record their working process. They used their own computers that had different window resolutions and color schema. As the task videos of 4 participants were corrupted, we used the 29 hours task videos of the 20 participants (8 from the first task and 12 from the second task) to evaluate our video scraping tool. Fig. 9 shows the box-plot of the Task Video Length (TVL in minutes) of these participants.

Based on the software tools that the participants used in our previous study, we defined application windows for the *scvRipper* tool to recognize Eclipse IDE window and web browser window (Google Chrome, Mozilla Firefox, Internet Explorer). Fig. 2 shows partially the definitions of the Eclipse IDE and Google Chrome window defined in this study. The definition instructs the *scvRipper* tool to scrap: 1) code editor and console view content in Eclipse IDE window, and 2) address bar, search box and web page content in web browser window (see Fig. 8 for an example).

B. Usefulness

Based on the time-series interaction data that the *scvRipper* tool extracted from the task videos, we performed two quantitative analysis of the participants' online search behavior during the two software development tasks. First, we computed a probabilistic model of the participants' search frequencies and intervals. Second, we studied the dynamics of the participants' working context over time.

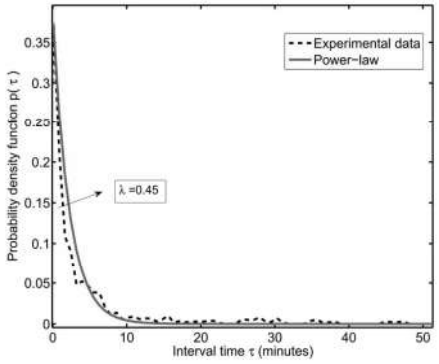


Fig. 10. The Distribution of Interval Time of Two Consecutive Queries

1) Search Frequencies and Intervals: The time-series interaction data identifies the search queries that the participants used during the tasks. We considered the first appearance of a search query in the time-series interaction data as the time when the participants searched the internet with this query. We collected the interval time of the two consecutive searches with different queries (denoted by τ) of the 20 developers in the two tasks. We used probability density function $p(\tau)$ to describe the relative likelihood of interval time of two consecutive searches between a given interval. We obtained probability density function of our data samples of interval time of two consecutive searches by kernel smoothing density estimation [43], as shown in black dot line in Fig. 10.

According to theory of human dynamics [44], the probability density function $p(\tau)$ of human activity interval time obeys a power-law distribution as $p(\tau) = k\lambda e^{-\lambda\tau}$, where λ is exponent parameter and k is a constant coefficient. We fitted our data samples of interval time of two consecutive searches in terms of this equation using Least Squares Fitting [45]. The fitting result is shown in red line in Fig. 10. This red

line is $p(\tau) = \frac{1}{1.41} \times 0.45e^{-0.45\tau}$. We employed coefficient of determination R^2 [46] to determine how well our experimental data fit the statistical model. The R^2 was 0.97 which indicates that our data samples can be well explained by the statistical model represented by the red line.

Given the probability density function $p(\tau)$, the probability of variable τ ranging from τ_1 to τ_2 is equal to $P(\tau_1 < \tau \leq \tau_2) = \int_{\tau_1}^{\tau_2} p(\tau) d\tau$ [47]. Based on the statistical model $p(\tau) = \frac{1}{1.41} \times 0.45e^{-0.45\tau}$, the probability that the developers in the two tasks searched with a different query within 1 minute is 0.48, within 3 minutes is 0.68, and within 10 minutes is 0.86.

2) The Dynamics of Working Context: The time-series interaction data identifies what software was used at different time periods. If the software used in the two consecutive time periods is Eclipse IDE and web browser respectively, we count one IDE $\rightleftharpoons$ Browser switching. If the software used in the two consecutive time periods is Eclipse (or web browser), we count one Within-IDE (or Within-Browser) switching.

Fig. 11 shows the number of IDE $\rightleftharpoons$ Browser switchings, Within-Browser switchings, and Within-IDE switchings that the developers performed in every 10 minutes in the two tasks. The box plots label data with 5 attributes. The bottom and top of the box are the first (25%) and third (75%) quartiles (Q_1 and Q_3) of the switchings that the developers performed in a 10-minute time slot. The band inside the box is the second quartile (Q_2, i.e., the median). The gray boxes indicate the interquartile range ($IQR = Q3 - Q1$). The lowest end of the whiskers represents minimal observation, and the highest end of whiskers represents maximal observation. The blue line shows the mean values of the number of switchings over time.

In the first task the developers started with a small number of IDE $\rightleftharpoons$ Browser switchings and a large number of Within-Browser switchings and Within-IDE switchings in the first 10 minutes. This indicates that the developers were trying to understand the problem they need to solve. Next, the developers' Within-Browser and Within-IDE switchings switchings remained relative stable or dropped in the 11-30 minutes, while the IDE $\rightleftharpoons$ Browser switchings increased in the 11-30 minutes. This indicates that the developers found good online examples and started integrating online examples in the IDE. Then, the developers' Within-Browser and IDE $\rightleftharpoons$ Browser switchings dropped for the rest of the first task, while the developers' Within-IDE switchings remained active. That is, the developers focused on developing the software within the IDE without much need for further online search.

In the second task the developers also started with a small number of IDE $\rightleftharpoons$ Browser switchings and a large number of Within-Browser switchings and Within-IDE switchings in the first 10 minutes. Next, there was a surge in the Within-Browser switchings in the 11-20 minutes followed by a surge in the IDE $\rightleftharpoons$ Browser switchings in 20-30 minutes. Similar to the first task, the developers found some useful online resources and started integrating them into the IDE in the first 30 minutes. However, the Within-Browser and IDE $\rightleftharpoons$ Browser switchings were much more intense in the second task than in the first task. Furthermore, the Within-Browser and IDE $\rightleftharpoons$ Browser switchings did not drop after the 30 minutes in the second task. Unlike the first task in which the developers' search activity occurred mainly in the beginning of the task, the developer int the second task had to frequently search and integrate online resources for the emerging problems throughout the task.

C. Runtime Performance

We ran our *scvRipper* tool on a Windows 7 computer with 4GB RAM and Intel(R) Core(TM)2 Duo CPU. The 29 hours task videos were recorded at sample rate 5 screenshots per second. As such, the 29 hours task videos consists of in total over 520K screenshots. Our *scvRipper* tool took 43 hours to identify about 11K distinct-content screenshots from the 29 hours videos at the threshold $t_{diff} = 0.7$. One distinct-content screenshot on average represents about 10 seconds video. The *scvRipper* tool took about 122 hours to extract time-series interaction data from the 11K distinct-content screenshots, i.e., on average 38.41 ± 16.94 seconds to analyze one distinct-content screenshot. The OCR of the scraped image content took about 60 hours .

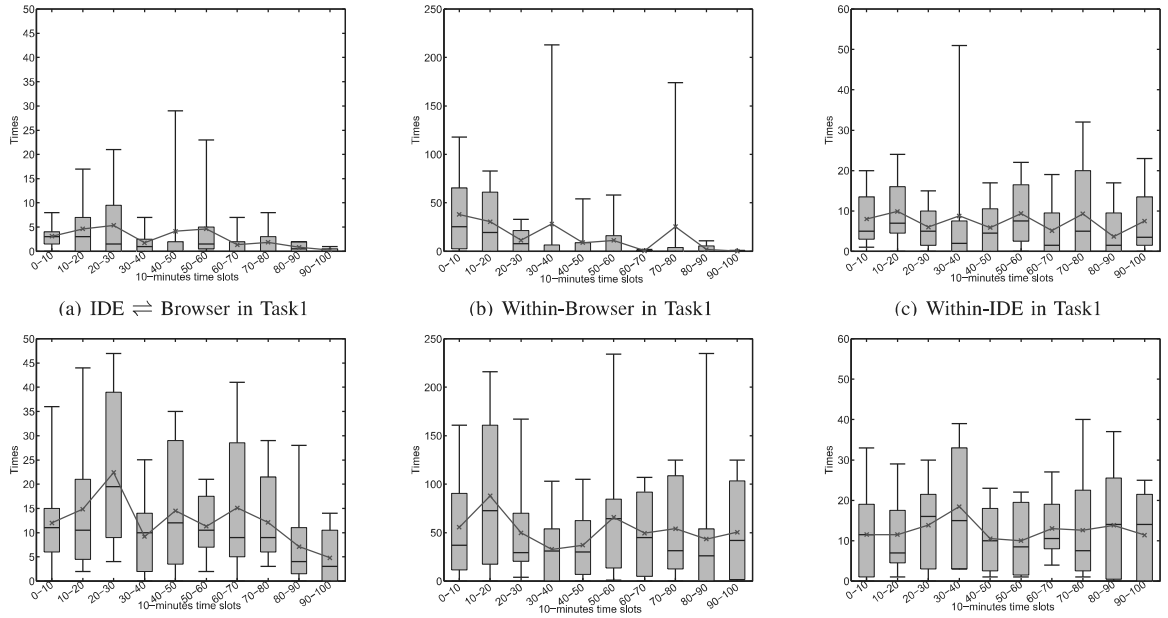
The current implementation of the *scvRipper*'s core algorithm processes one distinct-content screenshot at a time (i.e., sequential processing). The most time-consuming step of the core algorithm is the second step (i.e., detect individual visual cues). Our definition of the Eclipse IDE and Chrome window consists of about 30 and 20 visual cues respectively. The current implementation detects visual cues in a screenshot one at a time. This step consumes about 97% of the processing time of distinct-content screenshots. Since the processing of individual screenshots and the detection of individual visual cues are independent, the runtime performance of the *scvRipper* tool could be significantly improved by parallel computing [48] and hardware-implementation of template-matching algorithm [49]. Parallel computing and hardware acceleration [2] could also reduce the time of detecting distinct-content screenshots and the OCR of scraped screen images.

D. Effectiveness

We randomly sampled 500 distinct-content screenshots from different developers' task videos at different time periods. We qualitatively examined the screenshots that these sampled distinct-content screenshots represent. We found that the *scvRipper*'s image differencing technique (at $t_{diff} = 0.7$ in this study) can tolerate the reasonable differences between the screenshots caused by scrolling, mouse movement, and pop-up menus. Ignoring these screenshots should not cause significant information loss for data analysis.

We qualitatively examined the results of detected application windows in these sampled distinct-content screenshots. Our *scvRipper* tool sometimes may miss certain visual cues. As long as some visual cues were detected (over 80% of defined VisualCues in this study), *scvRipper* usually can still recognize the application window. However, missing some visual cues may result in the less accurate detection of window boundary.

[2]http://docs.opencv.org/modules/gpu/doc/introduction.html



(a) IDE ⇌ Browser in Task1 (b) Within-Browser in Task1 (c) Within-IDE in Task1

(d) IDE ⇌ Browser in Task2 (e) Within-Browser in Task2 (f) Within-IDE in Task2

Fig. 11. Statistics of Application and Content Switchings in Every 10 Minutes

For example, the detected window boundary may miss the title bar due to the failure of detecting the corresponding title bar visual cue. Our *scvRipper* tool can accurately recognize side-by-side or stacked windows. But it cannot accurately detect several ($\geq$ 3) overlapping windows, each of which is only partially visible. However, screenshots with several overlapping windows are rare in our dataset.

We evaluated the accuracy of the OCR results using the extracted query keywords. *scvRipper* identified 236 distinct-content screenshots that contain a search query. These queries contain 253 English words and 809 Chinese words in total. The OCR accuracy of the English words is about 88.5% (224/253), while the OCR accuracy of the Chinese words is about 74.9% (606/809). The screenshots had low DPI (Dots Per Inch, only 72-96 DPI in participants' computer) which is lower than the 300 DPI that the OCR tool generally requires. The OCR tool (ABBYY FineReader) we used scaled the low DIP screenshots to 300 DPI and produced acceptable OCR results.

VI. Related Work

Computer vision techniques have been used to identify user interface elements from screen-captured images or videos. Prefab [50] models widgets layout and appearance of an user interface toolkit as a library of prototypes. A prototype consists of a set of parts (e.g., a patch of pixels) and a set of constraints regarding those parts. Prefab identify the occurrence of widgets from a given prototype library in an image of an user interface by first assigning image pixels in parts from the prototype library and then filtering widget occurrences according to the part constraints.

Waken [51] uses image differencing technique to identify the occurrence of cursors, icons, menus, and tooltips that an application contains in screen-captured videos. The identified GUI elements can be associated with videos as metadata. This metadata allows the users to directly explore and interact with the video, as if it were a live application, for example, hove over icons in the video to display their associated tooltips.

Sikuli [52] uses template matching techniques [41] to find GUI patterns on the screen. It supports visual search of a given image in the screenshot. It also supports a visual scripting API to automate GUI interactions, for example automating GUI testing [53] or enhancing interactive help systems.

These computer-vision based techniques inspired the design and implementation of our video scraping technique, including the metamodel of application window, the detection of distinct-content screenshots, and the detection of application window. These existing techniques have focused on visual search, GUI automation, and implementing new interaction techniques. In contrast, our work focuses on reverse engineering time-series interaction data from screen-captured videos. Unlike the video data that only human can interpret, the reverse-engineered time-series interaction data can be automatically analyzed to discover behavioral patterns.

Instrumentation techniques [54, 55] can directly log a person's interaction with software tools and application content. They usually requires the support of sophisticated reflection APIs (e.g., Accessability API or UI Automation API) provided by applications, operating systems and GUI toolkits. Furthermore, a person can use several software tools (e.g., Eclipse IDE, different web browsers) in his work. Instrumenting all these software tools require significant efforts.

Some work proposes to combine low-level operating system APIs and computer vision techniques to track human computer interaction. Hurst et al. [15] leverages image differencing and template matching techniques to improve the accuracy of target identification that the users click. Chang et al. [16]

proposed a hybrid framework for detecting text blobs in user interface by combining pixel-based analysis and accessibility metadata of the user interface. In contrast, our video scrapping technique analyzes screen-captured videos without any accessibility information.

VII. Conclusions and Future Work

This paper presented a computer-vision based video-scraping technique (called *scvRipper*) that can automatically reverse-engineer time-series interaction data from screen-captured videos. This video-scraping technique is generic and easy to deploy. It can collect software usage and application content data across several applications according to the user's definition. Our *scvRipper* tool can address the increasing need for automatic observational data collection methods in the studies of human aspects of software engineering.

Our case study demonstrated the effectiveness and accuracy of the tool's reverse-engineered interaction data. It also demonstrated the usefulness of the extracted time-series interaction data in modeling ana analyzing the developers' online search behavior during software development. Our study also identified the bottleneck of the tool's runtime performance and suggested potential solutions.

We will improve the *scvRipper* tool's runtime performance using parallel computing and hardware acceleration. We are also interested in combining operating system level instrumentation (e.g., mouse and keystroke) with the core algorithm of *scvRipper* to collect more accurate time-series interaction data.

Acknowledgment

This research was supported by the National Basic Research Program of China (the 973 Program) under grant 2015CB352201, and National Key Technology R&D Program of the Ministry of Science and Technology of China under grant 2014BAH24F02. This work is supported by NTU SUG M4081029.020 and MOE AcRF Tier1 M4011165.020.

References

[1] A. von Mayrhauser and A. M. Vans, "Program understanding behavior during debugging of large scale software," in *Empirical Studies of Programmers, 7th Workshop*, pp. 157–179, ACM, 1997.

[2] C. L. Corritore and S. Wiedenbeck, "An exploratory study of program comprehension strategies of procedural and object-oriented programmers," *INT J HUM-COMPUT ST*, vol. 54, no. 1, pp. 1–23, 2001.

[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 Trans. Softw. Eng.*, vol. 39, no. 2, pp. 197–215, 2013.

[4] H. Li, Z. Xing, X. Peng, and W. Zhao, "What help do developers seek, when and how?," in *Proc. WCRE*, pp. 142–151, 2013.

[5] J. Wang, X. Peng, Z. Xing, and W. Zhao, "An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions," in *Proc. ICSM*, pp. 213–222, 2011.

[6] 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 Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, 2006.

[7] A. G. Koru, A. Ozok, and A. F. Norcio, "The effect of human memory organization on code reviews under different single and pair code reviewing scenarios," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1–3, 2005.

[8] P. Dewan, P. Agarwal, G. Shroff, and R. Hegde, "Distributed side-by-side programming," in *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, pp. 48–55, 2009.

[9] A. J. Ko and B. A. Myers, "A framework and methodology for studying the causes of software errors in programming systems," *J VISUAL LANG COMPUT*, vol. 16, no. 1, pp. 41–84, 2005.

[10] C. D. Hundhausen, J. L. Brown, S. Farley, and D. Skarpas, "A methodology for analyzing the temporal evolution of novice programs based on semantic components," in *Proceedings of the ACM International Computing Education Research Workshop*, pp. 59–71, 2006.

[11] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: An exploratory study," *IEEE Trans. Softw. Eng.*, vol. 30, no. 12, pp. 889–903, 2004.

[12] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar apis: An exploratory study," in *Proc. ICSE*, pp. 266–276, 2012.

[13] M. R. Leary, *Introduction to behavioral research methods*. Wadsworth Publishing Company, 1991.

[14] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *Prco. ICSE*, pp. 233–243, 2012.

[15] A. Hurst, S. E. Hudson, and J. Mankoff, "Automatically identifying targets users interact with during real world tasks," in *Proc. IUI*, pp. 11–20, ACM, 2010.

[16] T.-H. Chang, T. Yeh, and R. Miller, "Associating the visual representation of user interfaces with their internal structures and metadata," in *Proc. UIST*, pp. 245–256, 2011.

[17] J. Sillito, K. De Voider, B. Fisher, and G. Murphy, "Managing software change tasks: An exploratory study," in *International Symposium on Empirical Software Engineering*, pp. 10–pp, IEEE, 2005.

[18] C. L. Corritore and S. Wiedenbeck, "Direction and scope of comprehension-related activities by procedural and object-oriented programmers: An empirical study," in *Proc. IWPC*, pp. 139–148, IEEE, 2000.

[19] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector, "Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks," in *Proc. CHI*, pp. 1323–1332, ACM, 2008.

[20] D. Piorkowski, S. D. Fleming, C. Scaffidi, L. John, C. Bogart, B. E. John, M. Burnett, and R. Bellamy, "Modeling programmer navigation: A head-to-head empirical evaluation of predictive models," in *Proc. VL/HCC*, pp. 109–116, 2011.

[21] U. Dekel and J. D. Herbsleb, "Reading the documentation of invoked api functions in program comprehension," in *Proc. ICPC*, pp. 168–177, 2009.

[22] A. J. Ko, H. H. Aung, and B. A. Myers, "Design requirements for more flexible structured editors from a study of programmers' text editing," in *CHI'05 extended abstracts on human factors in computing systems*, pp. 1557–1560, ACM, 2005.

[23] A. J. Ko, H. H. Aung, and B. A. Myers, "Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks," in *Proc. ICSE*, pp. 126–135, 2005.

[24] M. Greiler, A. van Deursen, and M. Storey, "Test confessions: a study of testing practices for plug-in systems," in *Proc. ICSE*, pp. 244–254, 2012.

[25] E. R. Murphy-Hill, T. Zimmermann, and N. Nagappan, "Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development?," in *Proc. ICSE*, pp. 1–11, 2014.

[26] K. Brade, M. Guzdial, M. Steckel, and E. Soloway, "Whorf: A visualization tool for software maintenance," in *Proceedings 1992 IEEE Workshop on Visual Languages*, pp. 148–154, 1992.

[27] N. Ammar and M. Abi-Antoun, "Empirical evaluation of diagrams of the run-time structure for coding tasks," in *Proc. WCRE*, pp. 367–376, 2012.

[28] J. Lawrence, S. Clarke, M. Burnett, and G. Rothermel, "How well do professional developers test with code coverage visualizations? an empirical study," in *Proc. VL/HCC*, pp. 53–60, 2005.

[29] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, "Tesseract: Interactive visual exploration of socio-technical relationships in software development," in *Proc. ICSE*, pp. 23–33, 2009.

[30] M. W. Van Someren, Y. F. Barnard, J. A. Sandberg, *et al.*, *The think aloud method: A practical guide to modelling cognitive processes*, vol. 2. Academic Press London, 1994.

[31] A. J. Ko and B. A. Myers, "Designing the whyline: a debugging interface for asking questions about program behavior," in *Proc. CHI*, pp. 151–158, 2004.

[32] J. Wang, X. Peng, Z. Xing, and W. Zhao, "Improving feature location practice with multi-faceted interactive exploration," in *Proc. ICSE*, pp. 762–771, 2013.

[33] D.-C. Wu and W.-H. Tsai, "Spatial-domain image hiding using image differencing," *Proc. ICCVISP*, vol. 147, no. 1, pp. 29–37, 2000.

[34] J. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, no. 6, pp. 679–698, 1986.

[35] R. C. Gonzalez and R. E. Woods, "Digital image processing," 2002.

[36] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise.," in *Proc. Kdd*, vol. 96, pp. 226–231, 1996.

[37] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proc. ICCV*, vol. 2, pp. 1150–1157, 1999.

[38] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (surf)," *Computer vision and image understanding*, vol. 110, no. 3, pp. 346–359, 2008.

[39] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *Computer Vision–ECCV 2006*, pp. 430–443, Springer, 2006.

[40] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration.," in *VISAPP (1)*, pp. 331–340, 2009.

[41] D. A. Forsyth and J. Ponce, *Computer vision: a modern approach*. Prentice Hall Professional Technical Reference, 2002.

[42] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell*, vol. 22, no. 8, pp. 888–905, 2000.

[43] B. W. Silverman, *Density estimation for statistics and data analysis*, vol. 26. CRC press, 1986.

[44] A. L. Barabasi, "The origin of bursts and heavy tails in human dynamics," *Nature*, vol. 435, no. 7039, pp. 207–211, 2005.

[45] E. W. Weisstein, "Least squares fitting–exponential," *MathWorld-A Wolfram Web Resource. http://mathworld. wolfram. com/LeastSquaresFittingExponential. html*, 2011.

[46] A. Colin Cameron and F. A. Windmeijer, "An r-squared measure of goodness of fit for some common nonlinear regression models," *Journal of Econometrics*, vol. 77, no. 2, pp. 329–342, 1997.

[47] E. Parzen, "On estimation of a probability density function and mode," *Annals of mathematical statistics*, vol. 33, no. 3, pp. 1065–1076, 1962.

[48] Q. Zhang, Y. Chen, Y. Zhang, and Y. Xu, "Sift implementation and optimization for multi-core systems," in *Proc. IPDPS*, pp. 1–8, 2008.

[49] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, "Gpu-based video feature tracking and matching," in *EDGE, Workshop on Edge Computing Using New Commodity Architectures*, vol. 278, p. 4321, 2006.

[50] M. Dixon and J. Fogarty, "Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure," in *Proc. CHI*, pp. 1525–1534, 2010.

[51] N. Banovic, T. Grossman, J. Matejka, and G. Fitzmaurice, "Waken: reverse engineering usage information and interface structure from software videos," in *Proc. UIST*, pp. 83–92, 2012.

[52] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: using gui screenshots for search and automation," in *Proc. UIST*, pp. 183–192, 2009.

[53] T.-H. Chang, T. Yeh, and R. C. Miller, "Gui testing using computer vision," in *Proc. CHI*, pp. 1535–1544, 2010.

[54] D. M. Hilbert and D. F. Redmiles, "Extracting usability information from user interface events," *ACM Comput. Surv.*, vol. 32, no. 4, pp. 384–421, 2000.

[55] J. H. Kim, D. V. Gunn, E. Schuh, B. Phillips, R. J. Pagulayan, and D. Wixon, "Tracking real-time user experience (true): a comprehensive instrumentation solution for complex systems," in *Proc. CHI*, pp. 443–452, 2008.