

## Extracting and Analyzing Time-Series HCI Data from Screen-Captured Task Videos

Lingfeng Bao · Jing Li · Zhenchang Xing · Xinyu Wang · Bo Zhou

the date of receipt and acceptance should be inserted later

**Abstract** In recent years the amount of research on human aspects of software engineering has increased. Many studies use screencast software (e.g., Snagit) to record developers' behavior as they work on software development tasks. The recorded task videos capture direct information about which software tools the developers interact with and which content they access or generate during the task. Such Human-Computer Interaction (HCI) data can help researchers and practitioners understand and improve software engineering practices from human perspective. However, extracting time-series HCI data from screen-captured task 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 extract time-series HCI data from screen-captured videos. We have implemented our video scraping technique in a tool called *scvRipper*. We report the usefulness, effectiveness and runtime performance of the *scvRipper* tool using a case study of the 29 hours task videos of 20 developers in the two development tasks.

**Keywords** screen-captured video · video scraping · HCI data · online search behavior

---

L. Bao · X. Wang (✉) · B. Zhou  
College of Computer Science, Zhejiang University, Hangzhou, China  
E-mail: wangxinyu@zju.edu.cn

L. Bao  
E-mail: lingfengbao@zju.edu.cn

B. Zhou  
E-mail: bzhou@zju.edu.cn

J. Li · Z. Xing  
School of Computer Engineering, Nanyang Technological University, Singapore

J. Li  
E-mail: jli030@ntu.edu.sg

Z. Xing  
E-mail: zcxing@ntu.edu.sg

## 1 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 (von Mayrhauser and Vans 1997; Lawrance et al. 2013; Corritore and Wiedenbeck 2001), their information needs in developing and maintaining software (Wang et al. 2011; Ko et al. 2006; Li et al. 2013), how developers collaborate (Koru et al. 2005; Dewan et al. 2009), and what we can do to improve their performance (Ko and Myers 2005; Hundhausen et al. 2006; Robillard et al. 2004; Duala-Ekoko and Robillard 2012).

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 (Leary 1991). 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 screencast techniques. 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 (Vakilian et al. 2012). We refer to such data as *Human-Computer Interaction (HCI) data*. Instrumenting many of today's software systems is considerably complex. It often requires sophisticated reflection APIs (e.g., Accessibility API or UI Automation API) provided by applications, operating systems and GUI toolkits (Hurst et al. 2010; Chang et al. 2011). Furthermore, developers use various tools (e.g., IDE, web browsers) in software development tasks. Instrumenting all of these tools requires significant efforts.

Screencast techniques and tools offer a generic and easy-to-deploy alternative to instrumentation. Screencast software (e.g., Snagit<sup>1</sup>) can easily capture the developer's interaction with several software tools. It produces a screen-captured video, i.e., a sequence of time-ordered screenshots that the screencast tool takes at a given time interval (often 1/30 - 1/5 second). Each screenshot records the software tools the developer uses and the application content he accesses or generates at a specific time.

We surveyed 26 papers that were published in top-tier software engineering conferences from 1992 to 2014. These papers have studied various human aspects in software engineering. Screencast techniques were commonly used

---

<sup>1</sup> <http://www.techsmith.com/snagit.html>

to record the developers' behavior in these studies. However, many studies used video data mainly as qualitative evidence of study findings. Some studies (Lawrance et al. 2013; Li et al. 2013; Ko and Myers 2005) performed quantitative analysis of developers' behavior by manually transcribing and coding screen-captured videos into HCI data (e.g., software used, content accessed or generated). 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 to 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 and analyze the HCI 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 meet this need. Given a screen-captured video, our video scraping technique can recognize window-based applications in the screenshots of the video, and extract application content from the recognized application windows. It essentially transforms a screen-captured video into a time-series HCI data. A time-series HCI data consists of a sequence of time-ordered items. Each item captures the software tool(s) and application content shown on the screen in the screenshot at a specific time in the video.

We have implemented our video scraping technique in a video scraping tool called *scvRipper*. We conducted a case study to evaluate the usefulness, effectiveness, and runtime performance of our video scraping technique and the *scvRipper* tool. Our study demonstrated the effectiveness of video scraping technique in extracting time-series HCI data from screen-captured videos. Based on the extracted time-series HCI data, we conducted a quantitative analysis of the 20 developers' online search behavior in the two development tasks. This quantitative study demonstrated the usefulness of the time-series HCI data extracted from the screen-captured task videos for studying developers' behavior in software 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 2 summarizes our survey of the use of screen-captured videos in the 26 studies on human aspects of software engineering. Section 3 discusses a formative study of the challenges in the manual transcription of screen-captured videos. Section 4 discusses technical details of our video scraping technique. Section 5 reports our evaluation of the tool *scvRipper*. Section 6 reviews related work. Section 7 concludes the paper and discusses the future work.

## 2 A Survey of the Use of Screen-Capture Videos in SE Studies

We searched Google Scholar using keywords such as "software engineering", "exploratory study", "empirical study", "screencast" and/or "screen capture". From the search results, we surveyed 26 papers that studied human aspects

of software engineering. Among these 26 papers, 14 papers studied and modeled the developers' behavior in various software development tasks, such as debugging (von Mayrhauser and Vans 1997; Lawrance et al. 2013; Sillito et al. 2005), feature location (Wang et al. 2011), program comprehension (Corritore and Wiedenbeck 2001; Ko et al. 2006; Li et al. 2013; Robillard et al. 2004; Corritore and Wiedenbeck 2000; Lawrance et al. 2008; Piorkowski et al. 2011; Fritz et al. 2014), feature enhancement (Sillito et al. 2005), and using unfamiliar APIs (Duala-Ekoko and Robillard 2012; Dekel and Herbsleb 2009); 3 papers elicited information needs and requirements for improving software development tools (Ko and Myers 2005; Ko et al. 2005a,b); 5 papers studied software engineering practices such as novice programming (Hundhausen et al. 2006), pair programming (Koru et al. 2005), distributed programming (Dewan et al. 2009), testing of plugin systems (Greiler et al. 2012), and game development (Murphy-Hill et al. 2014); and 4 papers investigated visualization techniques of software data such as code structure (Brade et al. 1992; Ammar and Abi-Antoun 2012), program execution (Lawrence et al. 2005), and social relationship in software development (Sarma et al. 2009).

Our survey showed that screencast (also known as video screen capture) tools 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 (von Mayrhauser and Vans 1997; Lawrance et al. 2013; Koru et al. 2005; Piorkowski et al. 2011; Brade et al. 1992; Ammar and Abi-Antoun 2012; Sarma et al. 2009) used think-aloud technique (Van Someren et al. 1994) 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 (von Mayrhauser and Vans 1997; Koru et al. 2005; Lawrence et al. 2005) 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 (Leary 1991). Studying software engineering practices (e.g., peer programming (Koru et al. 2005), game development (Murphy-Hill et al. 2014), and plugin testing (Greiler et al. 2012)) usually used survey and interview methods that can collect only self-reported qualitative data.

Although screencast 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 video data into HCI data (e.g., software used, content accessed or generated) for the study purpose. 15 studies reported manual analysis of screen-captured videos in order to identify types of information the developers explored (Lawrance et al. 2013; Corritore and Wiedenbeck 2000; Lawrance et al. 2008), information foraging actions (Wang et al. 2011; Ko and Myers 2005; Hundhausen et al. 2006; Sillito et al. 2005; Dekel and Herbsleb 2009; Ko et al. 2005a,b), and patterns of developers' information behavior (Ko et al. 2006; Li et al. 2013; Robillard et al. 2004; Duala-Ekoko and Robillard 2012; Piorkowski et al. 2011). 4 papers (Ko

et al. 2006; Ko and Myers 2005; Hundhausen et al. 2006; Ko et al. 2005b) 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 HCI data to be collected.

The most costly studies were to study fine-grained behavioral patterns in software development tasks (e.g., (Wang et al. 2011; Ko and Myers 2005)) because they required iterative open coding of screen-captured videos. For example, Ko and Myers (Ko and Myers 2005) 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 (Ko and Myers 2004; Wang et al. 2013).

**Summary:** Previous human studies demonstrated the usefulness of screen-captured 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 automated tools that can extract and analyze time-series HCI data from screen-captured videos.

### 3 Formative Study

We conducted a formative study to better understand the challenges in manually transcribing screen-captured videos into time-series HCI data.

#### 3.1 Study Design

We recruited 3 graduate students from the School of Computer Engineering, Nanyang Technology University. We asked them to manually transcribe a 20-minutes screen-captured task video. The 20-minutes task video was excerpted from the 29-hours task videos that we collected in our previous field study of the developers’ online search behavior in software development tasks Li et al. (2013). The developers in that study used the Eclipse IDE to complete the two programming tasks. They used Chrome, Internet Explorer, and Firefox to search the Internet and browse web pages. In this formative study, we asked the participants to identify the applications that the developers used in the 20-minutes task video, the time and duration of application usage, and the application content that the developer interacted with (including source files viewed, web pages visited, and search queries issued).

We asked the participants to log their manual transcription results in a table like Table 1. A record in the table include the start *Time* of using an *Application* and the corresponding *Application Content*. For web browser, the application content has the URL of the web page currently visited and a query if the web page is a search engine result. For Eclipse, the application content

is the name of the source file currently viewed. The duration of application usage can be computed by subtracting start time of two consecutive records. The participants were also asked to identify application usage with unique content and assign it an unique *Index*. For example, the first web page visited is assigned *Url1*, the second web page is assigned *Url2*, the first source file viewed is assigned *Src1*, and so on. Note that the two web pages visited at Time 00:20 and 01:30 are the same. Thus, they are assigned the same Index *Url2*.

Table 1: An Example of Manual Transcription Logs

Index	Time	Application	Application Content
Url1	00:00	Chrome	URL: www.google.com Query:IPprogressMonitor editor
Url2	00:20	Chrome	URL: help.eclipse.org/...
Src1	01:05	Eclipse	SrcFile: MyEditor.java
Src2	01:10	Eclipse	SrcFile: SampleAction.java
Url2	01:30	Chrome	URL: help.eclipse.org/...

### 3.2 Results

Table 2 shows how much time each participant took to transcribe the 20-minutes task video, and how many records they logged. The results show that the ratio of video recording time to video analysis time is about 1:3-3.75. We can also see that the number of records that different participants logged are very different. The participant *S1* logged about 3 times more records than the participant *S3* did. We looked into the transcription results of the three participants. We found that the participant *S1* considered screenshots with different content resulting from window scrolling in the same web page or source file as different application content, while the participant *S1* and *S2* did not consider so. This results in much more records in *S1*'s transcription results than that of *S2* and *S3*. Furthermore, the participant *S3* omitted some application switchings whose duration was very short (i.e., switching from one application to another and then quickly switching back). This results in much less records in *S3*'s transcription results than that of *S1* and *S2*.

Table 2: The Statistics of Manual Transcription by the Three Participants

Participant	TotalTime (minute)	#NumOfRecords
S1	71	136
S2	56	73
S3	75	47

In the 20-minutes screen-captured video, the developer used two web search engines (Baidu and Google) and visited nine web pages. We further compared the search engines and web pages that the three participants logged. Table 3 shows the results. Note that different web pages from the same web site are annotated with an index number, such as topic.csdn.net (1), topic.csdn.net (2). We can see that the participant *S1* logged all the two search engines and the nine web pages, but both *S2* and *S3* missed three web pages. The missed web pages are from the same web site as some web pages previously visited. The participants *S2* and *S3* failed to recognize them.

Table 3: The Transcription Results of Search Engines or Web Pages Visited

Index	Search Engine or Web Page Visited	S1	S2	S3
1	www.baidu.com	✓	✓	✓
2	www.google.com.hk	✓	✓	✓
3	topic.csdn.net (1)	✓	✓	✓
4	topic.csdn.net (2)	✓	✗	✓
5	hongyegu.iteye.com	✓	✓	✓
6	www.itpub.net	✓	✓	✗
7	www.blogjava.net	✓	✓	✓
8	docs.oracle.com	✓	✓	✓
9	help.eclipse.org (1)	✓	✓	✓
10	help.eclipse.org (2)	✓	✗	✗
11	help.eclipse.org (3)	✓	✗	✗

Finally, we abstracted each record in the transcription results of the three participants into a universal identifier (UID) based on the query used, the web page visited, and the source file viewed in the record. As such, we obtained a sequence of UIDs for each participant. Note that the participant *S1* logged the same web page or source file with different content resulting from window scrolling as a sequence of records. As such, the sequence of UIDs of *S1* contains consecutive repetition of the same UID. We replaced the consecutive repetition of an UID with that UID. For example, a sequence of UIDs  $\{0, 0, 1, 1, 1, 2, 2, 1, 1\}$  will be replaced as the sequence  $\{0, 1, 2, 1\}$ . We computed the Longest Common Subsequence (LCS) of the sequence of UIDs of the two participants ( $S_i$  and  $S_j$ ). Then we measured the similarity of the sequence of UIDs of the two participants as  $\frac{2 * LCS}{|S_i| + |S_j|}$ , where  $|S_i|$  is the length of the sequence of UIDs of the participant  $S_i$ . As shown in Table 4, the transcription results of different participants overlap to certain extent, but the similarity of their transcription results is not high.

Table 4: The Similarity between the Sequence of UIDs of the Two Participants

	(S1, S2)	(S1, S3)	(S2, S3)
LCS	45	32	30
Similarity	0.62	0.46	0.67

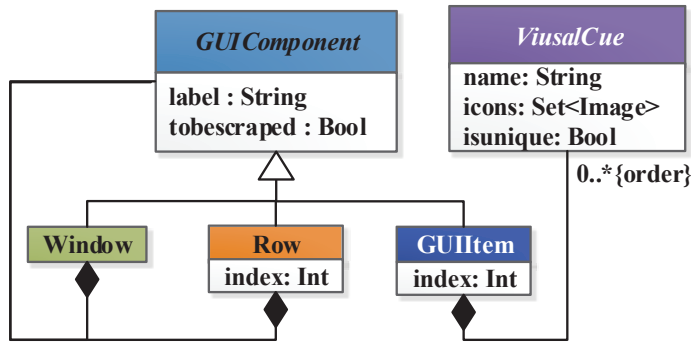


Fig. 1: The Metamodel of Application Windows

**Summary:** Our formative study shows that manual transcription of screen-captured videos requires significant time and effort. To obtain high-quality transcription results, a person must pay attention to micro-level of details. Due to the differences in observing and interpreting the videos, the transcription results by different participants can often be inconsistent.

#### 4 The Video Scraping Technique

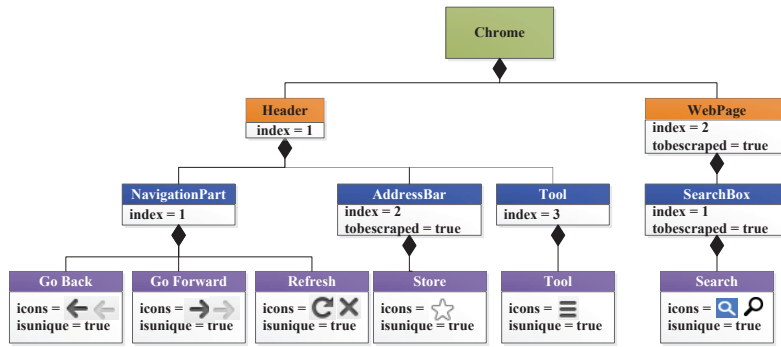
We now present our computer-vision based video scraping technique for automatically extracting time-series HCI data from screen-captured videos. We refer to our technique as *scvRipper*. In this section, we first describe the metamodel of application window *scvRipper* assumes. We then give overview of our *scvRipper* technique. Finally, we detail the key steps of *scvRipper*.

##### 4.1 Definition of Application Window

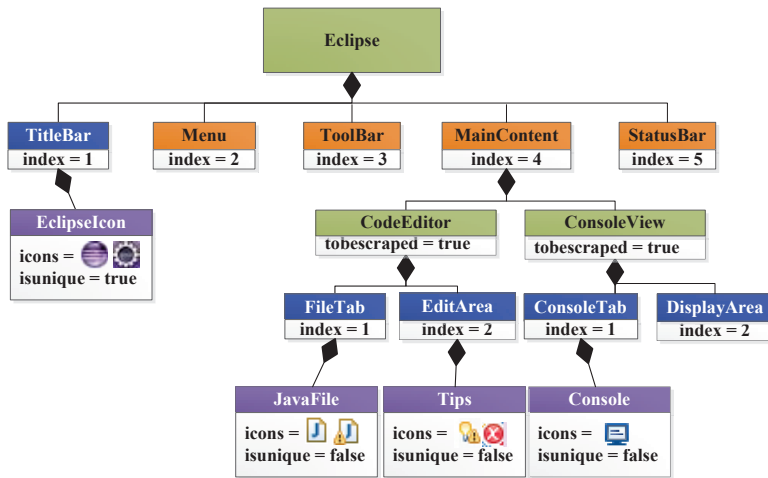
A person recognizes an application window based on his knowledge of the window layout and the distinct visual cues (e.g., icons) that appear in the window. Our video scraping technique (*scvRipper*) requires as input the definition of application windows to be recognized in the screenshots of a screen-captured 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.

Figure. 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





(a) Definition of Google Chrome Window



(b) Definition of Eclipse IDE Window

Fig. 2: Two Instances of Application-Window Metamodel

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

Figure. 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*

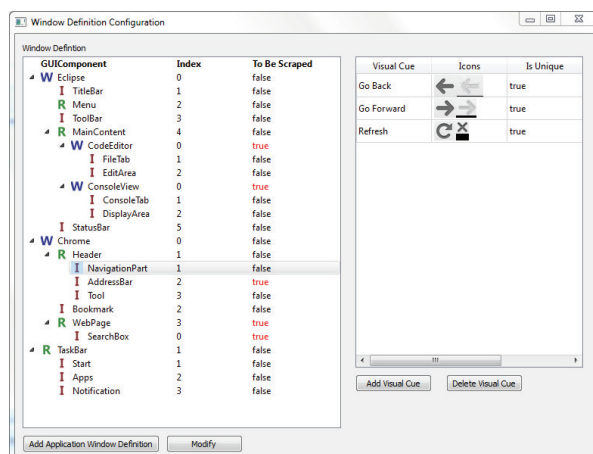


Fig. 3: The Configuration Tool for Window Definition

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 WebPage may contain a SearchBox GUIItem as commonly seen in search engine weppages. 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. Figure. 3 shows the screenshot of using configuration tool to define the Eclipse IDE window and the Google Chrome window shown in Figure. 2. 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.

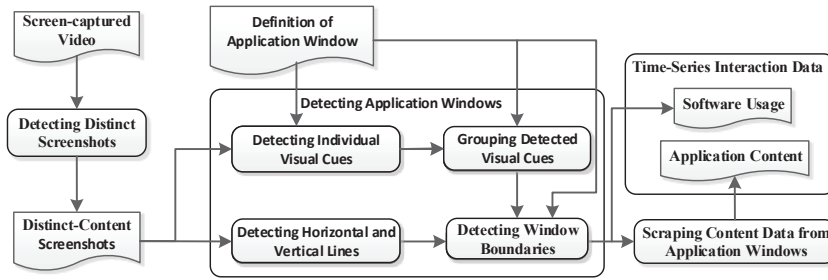


Fig. 4: The Process of Our Video Scraping Technique to Extract Time-Series HCI Data

#### 4.2 Technique Overview

Figure. 4 presents the process of our video scraping technique to extract time-series HCI data. 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 screenshots taken by screencast tools such as Snagit. It produces as output a time-series HCI data (i.e., software used and application content accessed/generated) extracted from the video. Our *scvRipper* tool essentially uses computer-vision techniques to transcribe a time-series screenshots that only human can interpret into a time-series HCI data that a computer can automatically analyze or mine for behavioral patterns.

First, *scvRipper* uses image differencing technique (Wu and Tsai 2000) 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.

The upper part of Figure. 5 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 Figure. 5 shows the time-series HCI data extracted from these four distinct-content screenshots according to the definition of Eclipse IDE and Google Chrome window in Figure. 2. Bulky contents (e.g., web page, code fragment) are omitted due to space limitation. This time-series HCI data identifies the software tools that the developer used at different time periods. It also identifies the application content that the

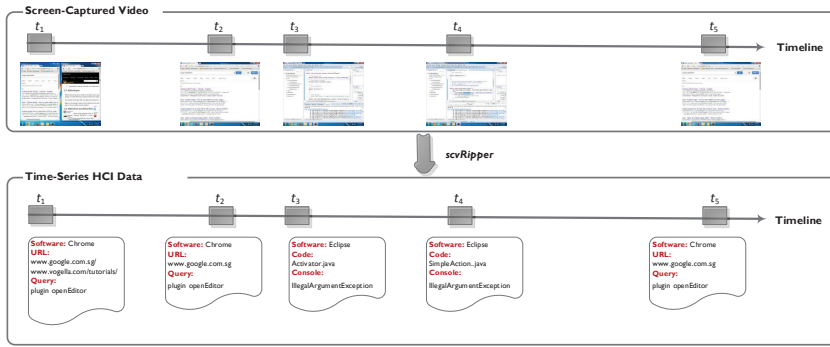


Fig. 5: An Illustrative Example of a Screen-Captured Video and Video Scrapping Results

developer accessed and/or generated (such as search queries, web pages, code fragments, and runtime exceptions) at different time periods.

In the following subsections, we describe technical details of our *scvRipper* technique.

#### 4.3 Detecting Distinct-Content Screenshots

The screencast tools can record a large number of screenshots (e.g., 30 screenshots per second). A sequence of consecutive screenshots can often 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 (Wu and Tsai 2000) to filter out subsequent screenshots with no or minor differences in the screen-captured video. This produces a sequence of distinct consecutive screenshots,  $s_1, s_2, \dots, s_n$  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 Figure. 5 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

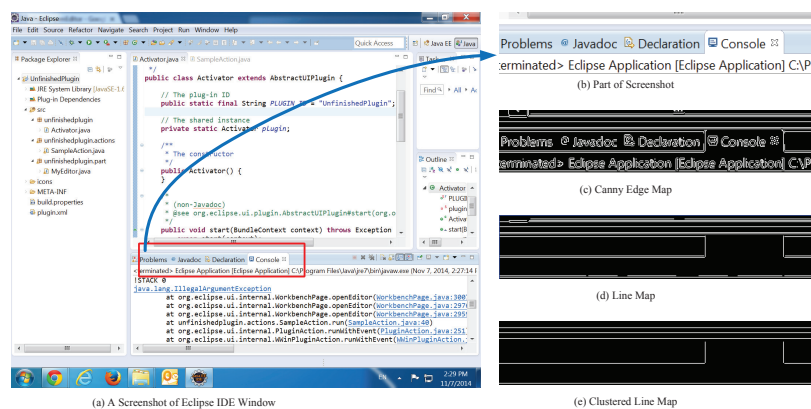


Fig. 6: An Example of Detecting Horizontal and Vertical Lines

120 seconds. A screencast tool can record 600 screenshots at the sample rate of 5 screenshots per second.

Given this stream of 600 screenshots, *scvRipper* can identify a sequence of five distinct consecutive screenshots as shown in Figure. 5. 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.

#### 4.4 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.

##### 4.4.1 Detecting Horizontal and Vertical Lines

Figure. 6 illustrates the process of detecting horizontal and vertical lines. Figure. 6(a) is the screenshot of the Eclipse window at time period  $t_3 - t_4$  in Figure. 5. *scvRipper* assumes that an application window (or subwindow) has explicit window boundaries and occupies a rectangular region in the screenshot. Thus, *scvRipper* first uses the canny edge detector (Canny 1986) 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. Figure. 6(c) shows the canny edge map of the part of the screenshot in Figure. 6(b).


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) (Gonzalez and Woods 2002) 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 erosion and dilation operations generates a line map of the screenshot (see Figure 6(d)).

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 (Ester et al. 1996)) 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.

#### 4.4.2 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 (Lowe 1999; Bay et al. 2008). 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 (Rosten and Drummond 2006) to detect the key points of an image. It extracts the Speeded Up Robust Features (SURF) (Bay et al. 2008) 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 (Muja and Lowe 2009). Figure. 7a visualizes the key points image of the part of the screenshot in Figure. 6(b). The left corner of Figure. 7b visualizes the key points image of the visual cue  of ConsoleView of Eclipse window. *scvRipper* detects the

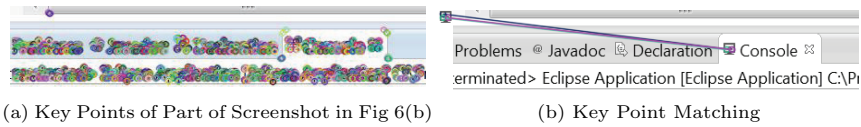



Fig. 7: An Example of Detecting Individual Visual Cues

occurrence of this visual cue in the screenshot as indicated by the lines in Figure. 7b

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 screenshot using template matching with alpha mask. The alpha mask of an image is a binary image used to reduce 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 (Forsyth and Ponce 2002). 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).

#### 4.4.3 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 window definition of the given application, *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 (Shi and Malik 2000) 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*

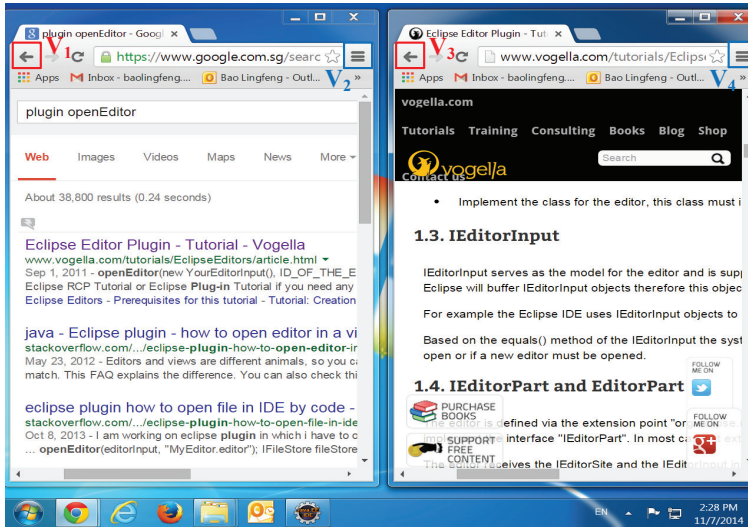


Fig. 8: An Example of Affinity Calculation

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. Figure. 8 shows the screenshot of the two side-by-side Chrome windows at time period  $t_1 - t_2$  in Figure. 5. 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  $V_4$ ).

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  $\delta$  is a term proportional to the image size. Intuitively, the more distance between the two visual cues, the less likely the two visual cues belong to the same application window. In Figure. 8 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  $V_4$ .

#### 4.4.4 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.

Figure. 9 shows the detected boundaries of the Eclipse window (at time period  $t_3 - t_4$  in Figure. 5) and the Chrome window (at time periods  $t_2 - t_3$  and  $t_5 - t_6$  in Figure. 5). 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 Figure. 1.

#### 4.5 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. Figure. 9 presents an example of the image scraping results of the Eclipse window and the Chrome window. The OCRed textual data records which contents the developer accesses or generates 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”.

### 5 Case Study

We used the 29-hours screen-captured task videos from our previous study (Li et al. 2013) to evaluate the runtime performance and effectiveness of our

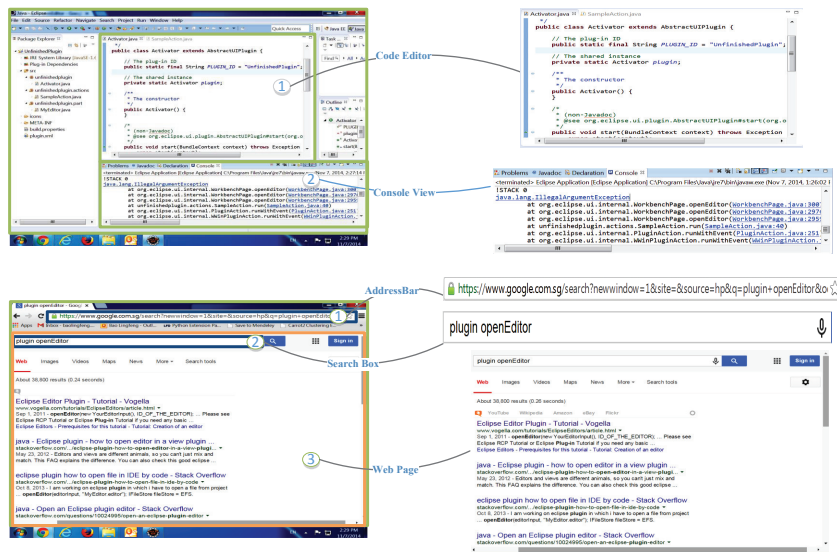


Fig. 9: An Example of Boundary Detection and Image Scraping Results

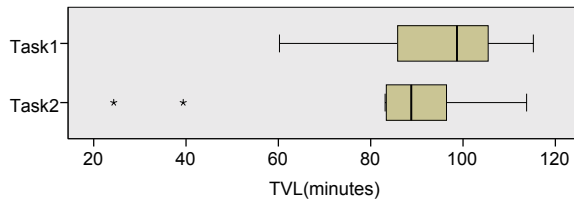


Fig. 10: The Statistics of Task Video Length (TVL)

*scuRipper* tool. Furthermore, we analyzed the developers' online search behavior in software development using the time-series HCI data extracted from the 29-hours videos. Finally, we discussed the findings in the developers' online search behavior and their implications for enhanced tool supports.

## 5.1 Data Set

The data we used is the screen-captured videos that we collected in our previous study of the developers' online search behavior during software development (Li et al. 2013). This previous study included two software development tasks. The first task (Task1) is to develop a new P2P chat software. The Task1 requires the knowledge about Java multi-threading, socket APIs, and GUI framework (e.g., Java Swing). The second task (Task2) is to maintain an existing Eclipse editor plugin. The Task2 includes two subtasks. The first subtask is to fix two bugs in the existing implementation. To fix these two bugs,

developers need knowledge about Eclipse editor API and plugin configuration. The second subtask asks developers to extend existing editor plugin with file open/save/close features and file content statistics (e.g., word count). This subtask requires developers to program to Eclipse editor and view extension points (e.g., `EditorPart`).

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. As the participants did not possess necessary knowledge for completing the tasks, they had to interleave coding, web search, and learning during the tasks.

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. 10 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. 9 for an example).

## 5.2 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 (about 50 screenshots). The *scvRipper* tool took about 122 hours to extract time-series HCI data from the 11K distinct-content screenshots, i.e., on average  $38.41 \pm 16.94$  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 (Zhang et al. 2008) and hardware-implementation of template-matching algorithm (Sinha et al. 2006). Parallel computing and hardware acceleration<sup>2</sup> could also reduce the time of detecting distinct-content screenshots and the OCR of scraped screen images.

### 5.3 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. 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.

### 5.4 Data Analysis on Online Search

This section analyzes the developers' online search activities using the extracted HCI data.

---

<sup>2</sup> <http://docs.opencv.org/modules/gpu/doc/introduction.html>

Table 5: The Top Three Most-Visited Web Sites of 7 Web Categories

	<b>The Top 3 Most-Visited Web Sites</b>
<b>Search engines (SE)</b>	www.google.com www.baidu.com www.bing.com
<b>Document sharing sites (DS)</b>	www.360doc.com www.doc88.com www.docin.com
<b>Technical tutorials (TT)</b>	blog.csdn.net www.newasp.cn developer.51cto.com
<b>Topic forums (TF)</b>	topic.csdn.net java.chinaitlab.com www.newsmth.net
<b>Code hosting sites (CH)</b>	download.csdn.net code.google.com github.com
<b>Q&amp;A sites (QA)</b>	zhidao.baidu.com stackoverflow.com iask.sina.com.cn
<b>API specification (API)</b>	docs.oracle.com developers.google.com www.aspose.com

#### 5.4.1 Most Visited Web Sites

First, we extracted web sites (i.e., domain name) from the scrapped URLs. We categorized the web sites that the developers visited during the two tasks into seven web categories: search engines (SE), technical tutorials (TT), document sharing sites (DS), topic forums (TF), code hosting sites (CH), Q&A sites (QA), and API specifications (API). Table 5 lists the top three most visited web sites of these seven categories in our study.

#### 5.4.2 Web Page Opened after a Search

Figure 11 presents the times that the developers opened a specific number of web pages after a search in the two tasks. We can see that the developers in the first task opened much less number of web pages after a search than the developer in the second task did. This reflects the complexity of the two tasks and the information needs of the developers in the two tasks.

#### 5.4.3 Web Page Visited and Web Page Switching

Figure 12 shows the number of unique URLs (i.e., web pages) that the 20 developers visited in the two tasks and the number of switchings between these web pages. In the first task 5 developers visited less than 9 web pages and made less than 9 web-page switchings. However, the other 3 developers visited on average  $20 \pm 4$  web pages and made on average  $37.6 \pm 19.5$  times web-page switchings. In the second task only 2 developers visited less than 6

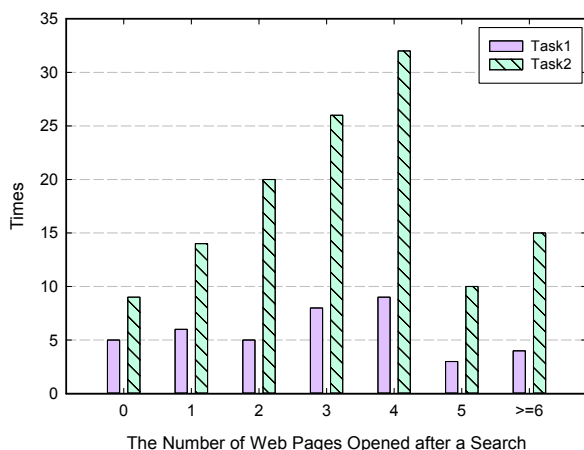


Fig. 11: The Statistics of Opening a Specific Number of Web Pages

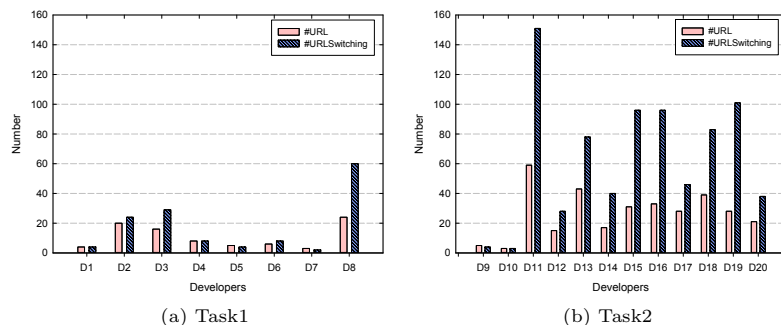


Fig. 12: Statistics of Unique URLs and URL Switchings

web pages and made less than 5 web-page switchings. These two developers are experienced Eclipse plugin developers. They completed the task much faster than the other 10 less experienced developers (i.e., the two “outlier” \* in Figure 10). During the task, they only issued a few searches and explored a small number of web pages. The other 10 developers visited on average  $31.4 \pm 13.1$  web pages and made on average  $75.7 \pm 38.1$  times web-page switchings.

The results show that in less than two hours of the tasks most of the developers had to explore, examine, and use a large amount of information when searching and using online resources. The developers often need to compare several web pages in order to select the web pages most relevant to the tasks. Depending on the developer’s working habit, this can sometimes result in the opening of a large number of web pages. For example, the developers D2 issued only 2 new queries. But he preferred to open several web pages from the search

results, and then examined these web pages to determine their relevance. He visited in total 20 web pages and made 24 times web-page switchings.

Furthermore, the developers often follow information scents from one web page to another during their online search process. This can also result in opening many web pages and switching between them. For example, the developer D15 searched “PermGen space”. He opened a web page about “PermGen space error” on “http://www.cnblogs.com” from the search results. He followed the link on this web page to another web page on “http://blog.csdn.net” about “JVM parameter setting”. This csdn web page helped him fix the virtual memory issue to run the Eclipse IDE.

Table 6: Statistics of Distinct Keywords and Keyword Sources

Developer ID	#DistinctKW	#FromCode	#FromConsole	#Self-phrasing
<b>Task1</b>				
D1	2	1	0	1
D2	5	0	0	5
D3	13	2	2	9
D4	12	0	0	12
D5	5	1	0	4
D6	13	0	0	13
D7	9	2	1	6
D8	13	1	0	12
mean±standard deviation(D1-D8)	9±4.15	0.87±0.78	0.37±0.69	7.75±4.11
<b>Task2</b>				
D9	10	2	0	8
D10	2	1	0	1
D11	32	14	6	12
D12	10	3	1	6
D13	9	1	0	8
D14	16	5	1	10
D15	13	1	5	7
D16	20	3	3	14
D17	18	5	0	13
D18	18	7	0	11
D19	9	5	0	4
D20	15	6		9
mean±standard deviation(D9-D20)	14.33±7.5	4.41±3.49	1.33±2.05	8.58±3.61

#### 5.4.4 Keyword Source in Queries

Given a search query extracted from search results web pages, we determined the sources of its keywords by searching code fragments and console outputs extracted from the distinct Eclipse IDE screenshots before the web-page screenshot in which a keyword was used for the first time. If a keyword appears in code fragments, for example, the keyword “openEditor” in the query “java.lang.IllegalArgumentException openEditor” is an Eclipse API used in the source code, we considered its source as “FromCode”. If a keyword appears in console outputs, for example, the keyword “IllegalargumentException” in the above query is an exception thrown in the console view, we considered its source as “FromConsole”. If a keyword appears in both code fragments and

Table 7: Most-Used Keywords in the Two Tasks

Keywords	Frequency (times)	Who Used These Keywords
<b>Task1</b>		
java	7	D1, D2, D3, D5, D6, D7, D8
socket	5	D1, D3, D4, D5, D7
TCP	4	D2, D3, D6, D8
SWT	3	D5, D6, D8
button	2	D3, D8
event	2	D5, D8
chat	2	D3, D6
<b>Task2</b>		
eclipse	10	D9, D11, D12, D13, D14, D15, D16, D17, D18, D20
plugin	8	D12, D13, D14, D15, D16, D17, D18, D20
EditorPart	6	D10, D11, D12, D17, D18, D19
openEditor	6	D12, D15, D16, D17, D18, D20
IEditorInput	5	D11, D13, D14, D19, D20
doSave	4	D11, D12, D18, D19
editor	4	D11, D12, D14, D16
IWorkbenchPage	4	D14, D17, D18, D20
SWT	4	D9, D11, D16, D17
savefile	4	D12, D13, D17, D20
view	4	D9, D11, D17, D20

console outputs, we consider the keyword as “FromCode”. If a keyword appears in neither code fragments nor console outputs, we considered its source as self-phrasing, for example, the keywords “eclipse” and “rcp” in the query “eclipse rcp EditorPart EdtorInput”.

Table 6 summarizes the number of distinct keywords that the developers used in the two tasks and the sources of these keywords. In the first task the developers keywords were mainly self-phrased. In the second task the keywords were both self-phrased and from IDE context.

Table 7 presents the top 7 most-used keywords by at least two developers in the first task and the top 11 most-used keywords by at least four developers in the second task. In the first task, all the seven most-used keywords were considered as self-phrasing. 3 of these 7 keywords were from task descriptions (socket, TCP, chat and 4 described programming language and techniques to be used (Java, SWT, button, event). Using these keywords the developers can find good online examples to complete the first task. They occasionally searched for unfamiliar APIs or errors (e.g., *IProgressMonitor* and *ConnectException*) while modifying reused code examples.

In the second task, 6 out of the 11 most-used keywords were considered as self-phrasing, three of which described application platform and techniques to be used (Eclipse, plugin, SWT) and three were from task description (editor, view, savefile). The other 5 most-used keywords were from IDE context, which described Eclipse APIs required for the task (*EditorPart*, *openEditor*, *IEditorInput*, *doSave*, *IWorkbenchPage*). In the second task, developers had to fix bugs of using specific Eclipse APIs and extend specific Eclipse interface. However, using only specific Eclipse APIs often cannot find good online examples to accomplish the second task. Developers had to use application and task context to restrict the search.



### 5.4.5 Query Refinement

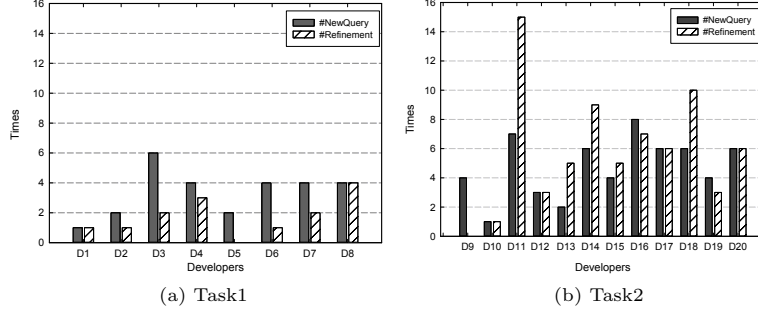


Fig. 13: The Statistics of New Queries and Query Refinements

Given the search queries that a developer issued, we measured the similarity of the two consecutive queries  $Q_i$  and  $Q_{i+1}$  using the Jaccard coefficient of distinct keywords of the two queries, i.e.,  $|Q_i \cap Q_{i+1}| / |Q_i \cup Q_{i+1}|$ . If the Jaccard coefficient of the two consecutive queries is greater than 0.5,  $Q_{i+1}$  is considered as the refinement of  $Q_i$ . For example, the 4th query “openEditor java.lanq.IllegalArqumentException” of the developer D11 was considered as a refinement of his 3rd query “java.lanq.IllegalArqumentException”, while his 5th query “eclipse rcv EditorPart EdtorInput” was considered as a new query as it was very different from the 4th query.

Figure 13 shows the number of new queries and the number of query refinements in the two tasks. In the first task the developers issued in total 27 new queries (on average  $3.37 \pm 1.34$  new queries per developer). The developers found satisfactory information in the search results of 7 of these 27 new queries and thus did not refine the queries. They refined 12 new queries 1-3 times, and 2 new queries more than 3 times. The rest 6 queries were too different from their preceding queries, and thus were considered as new queries. In the second task the developers issued in total 57 new queries (on average  $3.37 \pm 2.18$  new queries per developer). 9 of these 57 new queries were not refined because satisfactory information were found in the search results. 30 new queries were refined 1-3 times, and 9 new queries were refined more than 3 times. The rest 4 queries were considered as new queries because they were too different from their preceding queries.

### 5.4.6 Search Frequencies and Intervals

The extracted time-series HCI data identifies the search queries that the participants issued through the tasks. We considered the first appearance of a

search query in the time-series HCI 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  $\tau$ ) of the 20 developers in the two tasks. We used probability density function  $p(\tau)$  to describe the relative likelihood of the interval time of two consecutive searches between a given interval. We obtained the probability density function of our data samples of interval time of two consecutive searches by kernel smoothing density estimation (Silverman 1986), as shown in black dot line in Fig. 14.

According to theory of human dynamics (Barabasi 2005), 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  $\lambda$  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 (Weisstein 2011). The fitting result is shown in red line in Fig. 14. This red line is  $p(\tau) = \frac{1}{1.41} \times 0.45e^{-0.45\tau}$ . We employed coefficient of determination  $R^2$  (Colin Cameron and Windmeijer 1997) 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  $\tau$  ranging from  $\tau_1$  to  $\tau_2$  is equal to  $P(\tau_1 < \tau \leq \tau_2) = \int_{\tau_1}^{\tau_2} p(\tau)d\tau$  (Parzen 1962). 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.

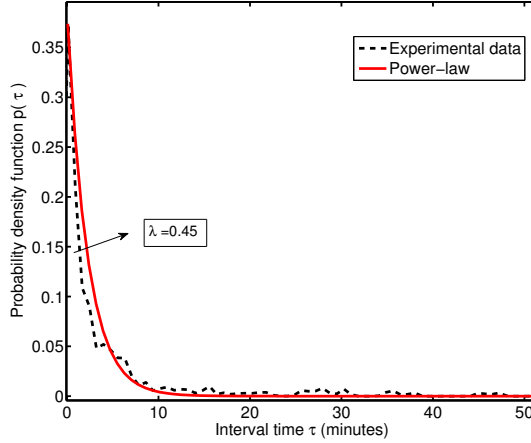


Fig. 14: The Distribution of Interval Time of Two Consecutive Queries

## 5.5 Data Analysis on Context Switching

This section analyzes the developers' context switching activities within and across the IDE and web browser.

### 5.5.1 Working Context Switching

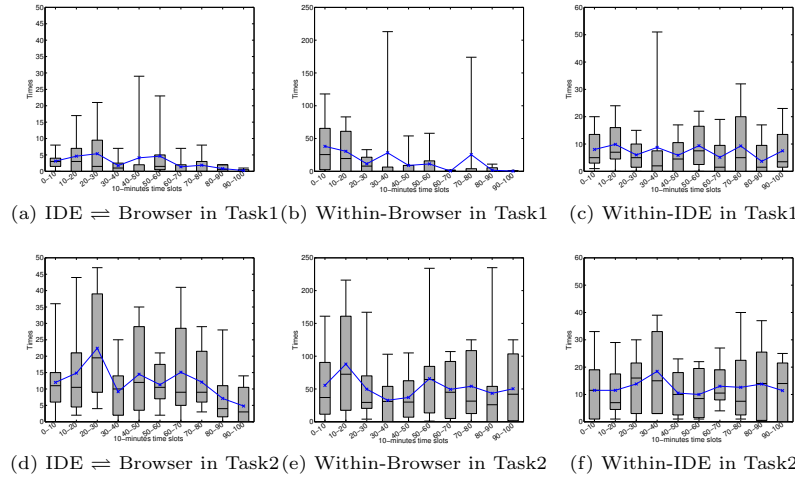


Fig. 15: Statistics of Application and Content Switchings in Every 10 Minutes

Given a sequence of distinct consecutive screenshots, if the two consecutive screenshots contain the Eclipse window and the web browser window respectively, we count one switching between IDE and web browser (IDE  $\rightleftharpoons$  Browser switching). If the two consecutive screenshots contain the same type of application windows (Eclipse IDE or web browser), we count one switching between distinct IDE content (Within-IDE switching) or one switching between distinct web content (Within-Browser switching). We also computed the time that the developers spent on the distinct IDE contents and the distinct web contents in the two tasks.

Fig. 15 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 ( $IQR = Q_3 - Q_1$ ). 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 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 in the second task had to frequently search and integrate online resources for the emerging problems throughout the task.

### 5.5.2 Markov Model on Context Switching

To further study the implicit information flow within web browser and between IDE and web browser, we built Markov Models (Whittaker and Poore 1993) for describing the developers' information flow behavior in Within-Browser switchings and in IDE  $\rightleftharpoons$  Browser switchings. The Markov Models consists of 8 states: the 7 web categories (see Table 5) and the Eclipse IDE. A transition between the two states represent the switching between the two web categories or the switching between a web category and the Eclipse IDE. The probability of a transition is computed based on the frequencies of the corresponding switchings, i.e., the number of switchings from one state to another state divided by the number of switchings from this state to all the states. Table 8 presents the the transition probabilities of the Markov Model. The maximal probability of each row is highlighted in bold font.

The table shows that the developers had the highest probabilities to switch between the Eclipse IDE and the technical tutorials (TT) in the first task. The technical tutorials seem to be the most useful information source in the first

Table 8: Markov Transition Matrices.

(a) Task 1

		Destination States							
		Eclipse	SE	TT	DS	TF	CH	QA	API
Source States	Eclipse	0	0.16	<b>0.62</b>	0.03	0.11	0	0.07	0.01
	SE	0.27	0	<b>0.34</b>	0.08	0.20	0.05	0.06	0
	TT	<b>0.73</b>	0.20	0	0	0.06	0	0.01	0
	DS	0.38	<b>0.50</b>	0	0	0.13	0	0	0
	TF	0.38	<b>0.45</b>	0.07	0	0	0	0.07	0.03
	CH	0.33	<b>0.67</b>	0	0	0	0	0	0
	QA	<b>0.42</b>	<b>0.42</b>	0.08	0	0.08	0	0	0
	API	<b>0.50</b>	<b>0.50</b>	0	0	0	0	0	0

(b) Task 2

		Destination States							
		Eclipse	SE	TT	DS	TF	CH	QA	API
Source States	Eclipse	0	0.12	0.26	<b>0.28</b>	0.15	0.02	0.01	0.15
	SE	0.19	0	0.26	0.06	0.14	0.03	0.05	<b>0.28</b>
	TT	<b>0.58</b>	0.25	0	0.03	0.05	0	0.02	0.07
	DS	<b>0.81</b>	0.09	0.04	0	0.02	0.02	0	0.02
	TF	<b>0.56</b>	0.25	0.08	0.02	0	0.01	0.01	0.08
	CH	<b>0.45</b>	0.31	0.03	0.07	0.03	0	0.07	0.03
	QA	0.20	<b>0.33</b>	0.10	0	0.10	0.13	0	0.13
	API	<b>0.53</b>	0.27	0.10	0.02	0.02	0.01	0.04	0

task. In fact, the technical tutorials often contain downloadable code examples that the developers can directly reuse to complete the task. In addition, the developer also integrated the information found on Q&A sites (QA) and API specification sites (API) in the IDE, as indicated by the high probabilities to switch from the QA or API categories to the Eclipse. In the first task, other than technical tutorials (TT), the developers had the highest probabilities to switch from different web categories (document sharing (DS), topic forum (TF), code hosting (CH), Q&A (QA), and API specification (API)) to the search engine. This suggests that the developers may collect hints from different web sites and then use the hints to refine their search.

The developers in the second task exhibited different information flow behavior. First, the probabilities to switch from the Eclipse IDE to different web categories (i.e., technical tutorials (TT), document sharing sites (DS), topic forums (TF), and API specifications (API)) are more evenly distributed. Furthermore, unlike the first task, the developers had highest probabilities to switch from technical tutorials (TT), document sharing (DS), topic forums (TF), code hosting (CH) sites, and API specifications (API) to the IDE, instead of to the search engine. This suggests that the technical tutorials were not the dominant information sources in the second task. The developers need more diverse information from different sources. Furthermore, the developers were more likely to integrate the information found on these information sources, instead of using the information to refine their search.

## 5.6 Implications of Behavior Analysis Results

Our analysis of time-series HCI data extracted from screen-captured task videos reveals the developers' micro-level behavioral patterns while they interleave coding and web search in software development. These micro-level behavioral patterns identify opportunities and challenges for supporting developers' online search during software development.

### *5.6.1 Context Sensing and Reasoning*

In light of previous work showing context to be useful in search tasks (Matejka et al. 2011; Brandt et al. 2010), our study suggests that more detailed studies are required to understand which types and scopes of context are more effective for providing useful results.

The scopes of context must be carefully determined. Existing tools use mainly the limited program context (e.g., a snapshot of current code) to augment the developer's queries. This limited context may not be sufficient to satisfy the developer's information needs in a task because the developer often needs application-level and task-level context to help to restrict the search. Application-level and task-level context may not be observable. Infer the high-level context (e.g., user interest) from the low-level observable contextual cues can be difficult. Spurious contextual information can introduce noise which may raise the rank of less-useful results. Showing contextual query keywords and allowing the developers to adjust them may be beneficial because it offers a mixed strategy to combine the developer's knowledge and the implicit context sensing and reasoning.

The dynamics of context must be carefully modeled. Many types of contextual information can be described as a static set of facts providing the background for online search. This static view of context may not be sufficient to reason about the developer's information needs over time because the developer's working context can change fast and frequently. Recommendation systems should avoid giving too many "helpful" hints by adjusting notification level based on the developer's progressions through the task. The developer's progression patterns may be modeled (e.g., using Hidden Markov Model (Rabiner and Juang 1986) or progression stages in time-evolving event sequences (Yang et al. 2014)) for predicting when the developer may most likely need online resources.

### *5.6.2 Exploratory Search of Online Resources*

The online search does not end with presenting a list of results. The developers have to explore, examine, and use many web pages and refine their queries in an iterative search process. This suggests that more intuitive presentation and interaction techniques are required to bridge the gulf of evaluation of online search results.

Our recent work (Wang et al. 2013) proposed an intelligent, multi-faceted, interactive search UI for exploring the feature location results in a code base. The automatically mined code facets provide to the developers more abstract and structured feedback about the feature location results. As a result, the developers can better refine their feature queries based on the hints they observe from different facets. This multi-faceted, exploratory search approach may also be beneficial for exploring multi-dimensional information space of online search results. Unlike source code, web contents vary greatly in formats as well as in both technical and presentational quality.

Entity-centric search (Bordes and Gabrilovich 2014; Guha et al. 2003) seems like a promising direction. Unlike current web search engines that essentially conduct page-level search, entity-centric search can uncover connected information about real-world entities. Entity-centric search has demonstrated its effectiveness in people search (Zhu et al. 2009), academic search (Nie and Zhang 2005), and product search (Nie et al. 2007). It can answer complex queries with direct and aggregate answers because of the availability of semantics defined by the knowledge graph (Nie and Zhang 2005; Bordes and Gabrilovich 2014). Otherwise, it could take one a long time to sift through many web pages returned by a page-level search engine. The challenge here is how to extract and model meaningful knowledge entities (e.g., programming languages, frameworks, application features) and their relationships from online software engineering resources.

### *5.6.3 Remembrance Agent and Community of Practices*

In searching and using online information, the developers' working context changes fast and frequently. The information flows implicitly during context switchings within and across applications. This suggests that effective techniques are required to track the information at micro-level and support smooth information flow as the developer interleaves coding and web search in software development.

Several tools (Brandt et al. 2010; Sawadsky and Murphy 2011; Ponzanelli et al. 2013) have been proposed to embed search engine or online resources into the IDE. These tools can reduce the switching cost of searching online resources while working in the IDE, especially when using online resources as reminders of technical details (Brandt et al. 2009). When the developers have to intensively search, browse, and learn online resources for a complex task, these tools may worsen the information overloading problem, because browsing several web pages in a small IDE view could be much less efficient than using normal web browser.

A remembrance agent (Rhodes 1996) would be useful to track the information that the developers search, browse and use during the task. Auto-completion technique could use the tracked information to augment human memory by displaying a list of information which may be relevant to the developer's current search or coding context. The tracked information further has the potential to support "community of practice" (Kimble et al. 2008;

Kushman and Katabi 2010; Matejka and Li 2009; Bateman et al. 2012; Hartmann et al. 2010). Our data analysis suggests the developers shared common information needs and information flow patterns in the task. The contextual “fingerprints” of some developer’s search history could be used to help other developers not only find relevant online resources in similar context, but also learn how to search for needed resources from others. According to theories of social learning (Bandura 1986) and cognitive apprenticeship (Brown et al. 1989), observing and imitating skilled practitioners performing the task in the context can help people incrementally adjust their performance until they reach competence.

## 6 Related Work

Computer vision techniques have been used to identify user interface elements from screen-captured images or videos. Prefab (Dixon and Fogarty 2010) 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 identifies 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 (Banovic et al. 2012) uses image differencing technique to identify the occurrence of GUI elements (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, hover over icons in the video to display their associated tooltips.

Sikuli (Yeh et al. 2009) uses template matching techniques (Forsyth and Ponce 2002) 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 (Chang et al. 2010) 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 extracting and analyzing time-series HCI data from screen-captured videos. Unlike the video data that only human can interpret, the extracted time-series HCI data can be automatically analyzed to discover behavioral patterns.

Instrumentation techniques (Hilbert and Redmiles 2000; Kim et al. 2008) can directly log a person’s interaction with software tools and application content. They usually requires the support of sophisticated reflection APIs (e.g., Accessibility API or UI Automation API) provided by applications, operat-



ing 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. (Hurst et al. 2010) leverages image differencing and template matching techniques to improve the accuracy of target identification that the users click. Chang et al. (Chang et al. 2011) 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.

## 7 Conclusions and Future Work

This paper presented a computer-vision based video-scrapping technique (called *scvRipper*) that can automatically extract time-series HCI data from screen-captured videos. This video-scrapping 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 extracted HCI data. It also demonstrated the usefulness of the extracted time-series HCI data in modeling and 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 HCI data.

## References

- N. Ammar, M. Abi-Antoun, Empirical Evaluation of Diagrams of the Run-time Structure for Coding Tasks, in *Proc. WCRE*, 2012, pp. 367–376
- A. Bandura, *Social foundations of thought and action: A social cognitive theory*, vol. 1 1986, p. 617
- N. Banovic, T. Grossman, J. Matejka, G. Fitzmaurice, Waken: reverse engineering usage information and interface structure from software videos, in *Proc. UIST*, 2012, pp. 83–92
- A.L. Barabasi, The origin of bursts and heavy tails in human dynamics. *Nature* **435**(7039), 207–211 (2005)
- S. Bateman, J. Teevan, R.W. White, The search dashboard: how reflection and comparison impact search behavior, in *Proc. CHI*, 2012, p. 1785
- H. Bay, A. Ess, T. Tuytelaars, L. Van Gool, Speeded-up robust features (surf). *Computer vision and image understanding* **110**(3), 346–359 (2008)
- A. Bordes, E. Gabrilovich, Constructing and Mining Web-scale Knowledge Graphs: KDD 2014 Tutorial, in *Proc. KDD*, 2014, p. 1967

- K. Brade, M. Guzdial, M. Steckel, E. Soloway, Whorf: A visualization tool for software maintenance, in *Proceedings 1992 IEEE Workshop on Visual Languages*, 1992, pp. 148–154
- J. Brandt, P.J. Guo, J. Lewenstein, M. Dontcheva, S.R. Klemmer, S. Francisco, Two Studies of Opportunistic Programming : Interleaving Web Foraging , Learning , and Writing Code, in *Proc. CHI*, 2009, pp. 1589–1598
- J. Brandt, M. Dontcheva, M. Weskamp, S.R. Klemmer, S. Francisco, Example-Centric Programming : Integrating Web Search into the Development Environment, in *Proc. CHI*, 2010, pp. 513–522
- J.S. Brown, A. Collins, P. Duguid, *Situated Cognition and the Culture of Learning*, 1989
- J. Canny, A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 679–698 (1986)
- T.-H. Chang, T. Yeh, R. Miller, Associating the visual representation of user interfaces with their internal structures and metadata, in *Proc. UIST*, 2011, pp. 245–256
- T.-H. Chang, T. Yeh, R.C. Miller, GUI testing using computer vision, in *Proc. CHI*, 2010, pp. 1535–1544
- A. Colin Cameron, F.A. Windmeijer, An r-squared measure of goodness of fit for some common nonlinear regression models. *Journal of Econometrics* **77**(2), 329–342 (1997)
- C.L. Corritore, S. Wiedenbeck, Direction and scope of comprehension-related activities by procedural and object-oriented programmers: An empirical study, in *Proc. IWPC*, IEEE, 2000, pp. 139–148. IEEE
- C.L. Corritore, S. Wiedenbeck, An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *INT J HUM-COMPUT ST* **54**(1), 1–23 (2001)
- U. Dekel, J.D. Herbsleb, Reading the documentation of invoked API functions in program comprehension, in *Proc. ICPC*, 2009, pp. 168–177
- P. Dewan, P. Agarwal, G. Shroff, R. Hegde, Distributed side-by-side programming, in *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, 2009, pp. 48–55
- M. Dixon, J. Fogarty, Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure, in *Proc. CHI*, 2010, pp. 1525–1534
- M. Dixon, D. Leventhal, J. Fogarty, Content and hierarchy in pixel-based methods for reverse engineering interface structure, in *Proc. CHI*, 2011, pp. 969–978
- E. Duala-Ekoko, M.P. Robillard, Asking and answering questions about unfamiliar APIs: An exploratory study, in *Proc. ICSE*, 2012, pp. 266–276
- J.S. Dumas, J. Redish, *A practical guide to usability testing* (Intellect Books, ???, 1999)
- M. El-Ramly, E. Stroulia, P. Sorenson, From run-time behavior to usage scenarios: An Interaction-Pattern Mining Approach, in *Proc. KDD*, 2002, p. 315
- M. Ester, H.-P. Kriegel, J. Sander, X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise., in *Proc. KDD*, vol. 96, 1996, pp. 226–231
- J. Fernquist, T. Grossman, G. Fitzmaurice, Sketch-sketch revolution: an engaging tutorial system for guided sketching and application learning, in *Proc. UIST*, 2011, pp. 373–382
- D.A. Forsyth, J. Ponce, *Computer vision: a modern approach* (Prentice Hall Professional Technical Reference, ???, 2002)
- T. Fritz, D.C. Shepherd, K. Kevic, W. Snipes, C. Bräunlich, Developers' code context models for change tasks. *Proc. FSE*, 7–18 (2014)
- R.C. Gonzalez, R.E. Woods, *Digital image processing* (Prentice hall Upper Saddle River, NJ, 2002)
- M. Greiler, A. van Deursen, M. Storey, Test confessions: a study of testing practices for plug-in systems, in *Proc. ICSE*, 2012, pp. 244–254
- T. Grossman, J. Matejka, G. Fitzmaurice, Chronicle: capture, exploration, and playback of document workflow histories, in *Proc. UIST*, 2010, pp. 143–152
- R. Guha, R. Guha, R. McCool, R. McCool, E. Miller, E. Miller, Semantic search, in *Proc. WWW*, 2003, pp. 700–709
- B. Hartmann, D. Macdougall, J. Brandt, S.R. Klemmer, What Would Other Programmers Do ? Suggesting Solutions to Error Messages, in *Proc. CHI*, 2010, pp. 1019–1028
- M.G. Helander, T.K. Landauer, P.V. Prabhu, *Handbook of human-computer interaction* (Elsevier, ???, 1997)

- D.M. Hilbert, D.F. Redmiles, Extracting usability information from user interface events. *ACM Comput. Surv.* **32**(4), 384–421 (2000)
- C.D. Hundhausen, J.L. Brown, S. Farley, 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*, 2006, pp. 59–71
- A. Hurst, S.E. Hudson, J. Mankoff, Automatically identifying targets users interact with during real world tasks, in *Proc. IUI*, ACM, 2010, pp. 11–20. ACM
- M.Y. Ivory, M.A. Hearst, The state of the art in automating usability evaluation of user interfaces. *ACM Comput. Surv.* **33**(4), 470–516 (2001)
- J.H. Kim, D.V. Gunn, E. Schuh, B. Phillips, R.J. Pagulayan, D. Wixon, Tracking real-time user experience (TRUE): a comprehensive instrumentation solution for complex systems, in *Proc. CHI*, 2008, pp. 443–452
- C. Kimble, P.M. Hildreth, I. Bourdon, *Communities of Practice: Creating Learning Environments for Educators* vol. v. 1 (Information Age Pub., ???, 2008)
- A.J. Ko, B.A. Myers, Designing the whyline: a debugging interface for asking questions about program behavior, in *Proc. CHI*, 2004, pp. 151–158
- A.J. Ko, B.A. Myers, A framework and methodology for studying the causes of software errors in programming systems. *J VISUAL LANG COMPUT* **16**(1), 41–84 (2005)
- A.J. Ko, H.H. Aung, 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*, ACM, 2005a, pp. 1557–1560. ACM
- A.J. Ko, H.H. Aung, B.A. Myers, Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks, in *Proc. ICSE*, 2005b, pp. 126–135
- A.J. Ko, B.A. Myers, M.J. Coblenz, H.H. Aung, An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.* **32**(12), 971–987 (2006)
- A.G. Koru, A. Ozok, 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, 2005, pp. 1–3
- N. Kushman, D. Katabi, Enabling Configuration-Independent Automation by Non-Expert Users. *Proceedings of the Ninth USENIX Symposium on Operating Systems Design and Implementation*, 223–236 (2010)
- J. Lawrance, R. Bellamy, M. Burnett, K. Rector, Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks, in *Proc. CHI*, ACM, 2008, pp. 1323–1332. ACM
- J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, S.D. Fleming, How programmers debug, revisited: An information foraging theory perspective. *IEEE Trans. Softw. Eng.* **39**(2), 197–215 (2013)
- J. Lawrence, S. Clarke, M. Burnett, G. Rothermel, How well do professional developers test with code coverage visualizations? An empirical study, in *Proc. VL/HCC*, 2005, pp. 53–60
- M.R. Leary, *Introduction to behavioral research methods* (Wadsworth Publishing Company, ???, 1991)
- V.I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady* **10**(8), 707–710 (1966)
- H. Li, Z. Xing, X. Peng, W. Zhao, What help do developers seek, when and how?, in *Proc. WCRE*, 2013, pp. 142–151
- D.G. Lowe, Object recognition from local scale-invariant features, in *Proc. ICCV*, vol. 2, 1999, pp. 1150–1157
- J. Matejka, W. Li, CommunityCommands: Command Recommendations for Software Applications. *Proc. UIST*, 193–202 (2009)
- J. Matejka, T. Grossman, G. Fitzmaurice, Ambient help, in *Proc. CHI*, 2011, pp. 2751–2760
- M. Muja, D.G. Lowe, Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration., in *VISAPP (1)*, 2009, pp. 331–340
- E.R. Murphy-Hill, T. Zimmermann, N. Nagappan, Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development?, in *Proc. ICSE*, 2014, pp. 1–11

- T. Nakamura, T. Igarashi, An application-independent system for visualizing user operation history, in *Proc. UIST*, 2008, pp. 23–32
- Z. Nie, Y. Zhang, Object-Level Ranking : Bringing Order to Web Objects, in *Proc. WWW*, 2005, pp. 567–574
- Z. Nie, Y. Ma, S. Shi, J.-r. Wen, W.-y. Ma, Web Object Retrieval, in *Proc. WWW*, 2007, pp. 81–90
- E. Parzen, On estimation of a probability density function and mode. *Annals of mathematical statistics* **33**(3), 1065–1076 (1962)
- D. Piorkowski, S.D. Fleming, C. Scaffidi, L. John, C. Bogart, B.E. John, M. Burnett, R. Bellamy, Modeling programmer navigation: A head-to-head empirical evaluation of predictive models, in *Proc. VL/HCC*, 2011, pp. 109–116
- L. Ponzanelli, A. Bacchelli, M. Lanza, Seahawk: Stack overflow in the IDE, in *Proc. ICSE*, 2013, pp. 1295–1298
- L. Rabiner, B.H. Juang, An introduction to hidden markov models. *IEEE ASSP Magazine* **3**(1), 4–16 (1986)
- R. Ragulayan, D. Gunn, R. Romero, *A gameplay-centered design framework for human factors in games* (Taylor & Francis, 2006)
- L.M. Rea, R.A. Parker, *Designing and conducting survey research: A comprehensive guide* (Proc. John Wiley & Sons, ???, 2012)
- B. Rhodes, Remembrance Agent: A continuously running automated information retrieval system, in *The Proceedings of The First International Conference on The Practical Application Of Intelligent Agents and Multi Agent Technology*, 1996, pp. 122–125
- M.P. Robillard, W. Coelho, G.C. Murphy, How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.* **30**(12), 889–903 (2004)
- E. Rosten, T. Drummond, Machine learning for high-speed corner detection, in *Computer Vision–ECCV 2006* (Springer, ???, 2006), pp. 430–443
- A. Sarma, L. Maccherone, P. Wagstrom, J. Herbsleb, Tesseract: Interactive visual exploration of socio-technical relationships in software development, in *Proc. ICSE*, 2009, pp. 23–33
- N. Sawadsky, G.C. Murphy, Fishtail: From Task Context to Source Code Examples, in *Proceeding of the 1st workshop on Developing tools as plug-ins - TOPI*, 2011, p. 48
- J. Shi, J. Malik, Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell* **22**(8), 888–905 (2000)
- J. Sillito, K. De Volder, B. Fisher, G. Murphy, Managing software change tasks: An exploratory study, in *International Symposium on Empirical Software Engineering*, IEEE, 2005, p. 10. IEEE
- B.W. Silverman, *Density estimation for statistics and data analysis*, vol. 26 (CRC press, ???, 1986)
- S.N. Sinha, J.-M. Frahm, M. Pollefeys, Y. Genc, GPU-based video feature tracking and matching, in *EDGE, Workshop on Edge Computing Using New Commodity Architectures*, vol. 278, 2006, p. 4321
- M. Vakilian, N. Chen, S. Negara, B.A. Rajkumar, B.P. Bailey, R.E. Johnson, Use, disuse, and misuse of automated refactorings, in *Proc. ICSE*, 2012, pp. 233–243
- 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)
- A. von Mayrhauser, A.M. Vans, Program understanding behavior during debugging of large scale software, in *Empirical Studies of Programmers, 7th Workshop*, ACM, 1997, pp. 157–179. ACM
- J. Wang, X. Peng, Z. Xing, W. Zhao, An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions, in *Proc. ICSM*, 2011, pp. 213–222
- J. Wang, X. Peng, Z. Xing, W. Zhao, Improving feature location practice with multi-faceted interactive exploration, in *Proc. ICSE*, 2013, pp. 762–771
- M. Weigel, V. Mehta, J. Steimle, More than touch: understanding how people use skin as an input surface for mobile computing, in *Proc. CHI*, ACM, 2014, pp. 179–188. ACM
- E.W. Weisstein, Least squares fitting–exponential. MathWorld–A Wolfram Web Resource. <http://mathworld.wolfram.com/LeastSquaresFittingExponential.html> (2011)
- J.A. Whittaker, J.H. Poore, *Markov analysis of software specifications*, 1993

- 
- D.-C. Wu, W.-H. Tsai, Spatial-domain image hiding using image differencing. *Proc. IC-CVISP* **147**(1), 29–37 (2000)
- J. Yang, J. McAuley, J. Leskovec, P. LePendou, N. Shah, Finding progression stages in time-evolving event sequences, in *Proc. WWW*, 2014, pp. 783–794
- T. Yeh, T.-H. Chang, R.C. Miller, Sikuli: using GUI screenshots for search and automation, in *Proc. UIST*, 2009, pp. 183–192
- Q. Zhang, Y. Chen, Y. Zhang, Y. Xu, SIFT implementation and optimization for multi-core systems, in *Proc. IPDPS*, 2008, pp. 1–8
- J. Zhu, Z. Nie, X. Liu, B. Zhang, J.-R. Wen, StatSnowball: a statistical approach to extracting entity relationships, in *Proc. WWW*, 2009, p. 101