Empir Software Eng
DOI 10.1007/s10664-015-9417-1

Extracting and analyzing time-series HCI data from screen-captured task videos

Lingfeng Bao[1] · Jing Li[2] · Zhenchang Xing[2] · Xinyu Wang[1] · Xin Xia[1] · Bo Zhou[1]

Abstract Recent years have witnessed the increasing emphasis on human aspects in software engineering research and practices. Our survey of existing studies on human aspects in software engineering shows that screen-captured videos have been widely used to record developers' behavior and study software engineering practices. The screen-captured videos provide 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 report a formative study to understand the challenges in manually transcribing screen-captured videos into time-series HCI data. We then

Communicated by: Emerson Murphy-Hill

✉ Xinyu Wang
wangxinyu@zju.edu.cn

Lingfeng Bao
lingfengbao@zju.edu.cn

Jing Li
jli030@ntu.edu.sg

Zhenchang Xing
zcxing@ntu.edu.sg

Xin Xia
xxia@zju.edu.cn

Bo Zhou
bzhou@zju.edu.cn

[1] College of Computer Science, Zhejiang University, Hangzhou, China

[2] School of Computer Engineering, Nanyang Technological University, Singapore, Singapore

present a computer-vision based video scraping technique to automatically extract time-series HCI data from screen-captured videos. We also present a case study of our *scvRipper* tool that implements the video scraping technique using 29-hours of task videos of 20 developers in two development tasks. The case study not only evaluates the runtime performance and robustness of the tool, but also performs a detailed quantitative analysis of the tool's ability to extract time-series HCI data from screen-captured task videos. We also study the developer's micro-level behavior patterns in software development from the quantitative analysis.

Keywords Screen-captured video · Video scraping · HCI data · Online search behavior

1 Introduction

It has long been recognized that the humans involved in software engineering, including the developers and other stakeholders, are a key factor in determining project outcomes and success. A number of workshops and conferences (e.g. CHASE, VL/HCC) have been focusing on the human and social aspects in software engineering. Some important objectives of these studies are 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).

Unlike 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 (also known as video screen capture) techniques. We can instrument software tools that the developers use to log their interaction with the tools and application content. For example, Eclipse IDE can record which refactorings a developer applied 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 effort.

Screencast techniques and tools offer a generic and easy-to-deploy alternative to instrumentation. Screencast software (e.g., Snagit[1]) can easily capture a developer's interaction

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

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 specified 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 survey 26 papers that are published in top-tier software engineering conferences from 1992 to 2014. These papers have studied various human aspects in software engineering. Screencast techniques are commonly used to record developers' behavior in these studies. However, many studies use video data mainly as qualitative evidence of study findings. Some studies (Lawrance et al. 2013; Li et al. 2013; Ko and Myers 2005) perform 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 provide deeper insight into the developers' behavior in various software development tasks. Such quantitative analysis is expensive and time consuming. It is reported that the ratio of video recording time to video analysis time is about 1:4–7.

As software engineering research has increased focus on human aspects, there has been a greater need for a solution to automatically extract and analyzing the HCI data from screen-captured videos, in order to facilitate quantitative analysis of the developers' behavior in software development tasks. To meet this need, we design and implement a computer-vision-based video scraping technique called *scvRipper* (Bao et al. 2015b, c). Given a screen-captured video, *scvRipper* can recognize window-based applications in the screenshots of video, and extract application content from the recognized application windows. It essentially transforms a screen-captured video into a time-series HCI data. 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. The application content contains the working information of an application at a specific time, such as URL in an address bar, query in a search box, and web page content in a web browser, and code fragment in a code editor and console output in Eclipse IDE.

In addition to the screen-scraping approach and the *scvRipper* tool (Bao et al. 2015b, c), we make the following additional contributions in this paper:

– We complement our previous work (Bao et al. 2015b, c) with a formative study in order to better understand the challenges in manually transcribing screen-captured videos. Our study shows that manual transcription requires constant attention to micro-level details in the videos.
– We conduct a detailed quantitative analysis of the *scvRipper* tool's ability to extract fine-grained time-series HCI data from screen-captured videos. The study is based on 29-hour screen-captured task videos of 20 developers in the two development tasks. This quantitative study demonstrates the usefulness of the *scvRipper* tool for studying developers' micro-level behavior in software development tasks.

The rest of this 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 scrapping technique. Section 5 reports our evaluation of the *scvRipper* tool. Section 6 discusses the potential applicability and limitation of our *scvRipper* technique in studying human factors of software engineering. Section 7 reviews related work. Section 8 concludes the paper and discusses our future plan.

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

We search using Google Scholar for keywords such as “software engineering”, “exploratory study”, “empirical study”, “screencast” and/or “screen capture” during the period 2–10 November 2014. From the search results, we survey 26 papers that study human aspects of software engineering (see Table 1).

We categorize these 26 papers into 4 different kinds according to their research purposes:

1. To study and model the developers’ behavior in software development tasks, such as debugging, feature location, program comprehension, using unfamiliar APIs.
2. To elicit information needs and requirements for improving the design of software development tools.
3. To study software engineering practices, such as novice programming, pair programming, distributed programming, testing of plugin systems, game development.
4. To investigate visualization techniques, like code structure, program execution, social relationships in software development

We further investigate how these studies analyze the screen-captured videos. We identify 3 video coding levels in these studies:

– Artifacts: Refers to information developers access or generate. For example, (Lawrance et al. 2013) categorized several artifacts such as “Bug-text”, “Runtime”, “Source Code”, “UI-static inspection”, “Web” in their study of debugging activities. Corritore and Wiedenbeck (2001, 2000) identified files that the participants accessed. Li et al. (2013) extracted online resources (e.g. blogs, API documents) that developers used from screen-captured videos.
– Actions: Refers to developers’ action during software development tasks, such as reading and editing code, navigation, searching, switching between applications, switching between documents, and testing. In the surveyed papers, the researchers focus on different actions according to their study objectives. For example, (Piorkowski et al. 2011; Fritz et al. 2014; Robillard et al. 2004; Ammar and Abi-Antoun 2012) only cared about the navigation actions in their studies, while more action types (e.g. reading, searching, switching) were considered in many other studies (Sillito et al. 2005; Wang et al. 2011; Ko et al. 2006; Duala-Ekoko and Robillard 2012).
– Qualitative: Refers to qualitative analysis using screen-captured videos. For example, to validate developers’ intentions and strategies during the task. Qualitative analysis is usually based on the analyst’s subjective summary of his observation, and thus does not require detailed information about artifacts and developer actions.

Our survey shows that screencast tools have been widely used to collect observational data in studying human aspects of software engineering (21 of 26 papers), especially for modeling developers’ behavior in software development tasks (research purpose 1) and eliciting design requirements for innovative software development tools (research purpose 2). Studies with these two research purposes require detailed information about developers’ actions and the artifacts developers use in software development tasks. Studying software engineering practices (research purpose 3) usually requires only qualitative data, such as the developers’ intentions and strategies. As such, think-aloud, survey and interview methods are commonly used in these studies. Investigation of visualization techniques usually uses both screen-captured videos and other data collection methods. However, screen-captured videos are analyzed qualitatively without the need for detailed user actions and artifacts.

Table 1 The overview of surveyed papers

Paper	Screen-captured Video[1]	Other Collection Method[2]	Research Purpose [3]	Video Coding Level
von Mayrhauser and Vans (1997)	×	Video tape Audio Think-aloud	1-debugging	Qualitative
Lawrance et al. (2013)	✓	Audio Think-aloud	1-debugging	Artifacts
Sillito et al. (2005)	✓	Audio Log	1-debugging	Actions
Wang et al. (2011)	✓	N.A.	1-feature location	Actions
Corritore and Wiedenbeck (2001)	✓	N.A.	1-program comprehension	Artifacts
Ko et al. (2006)	✓	N.A.	1-program comprehension	Actions
(Li et al. 2013)	✓	N.A.	1-program comprehension	Artifacts & Actions
Robillard et al. (2004)	✓	N.A.	1-program comprehension	Actions
Corritore and Wiedenbeck (2000)	✓	N.A.	1-program comprehension	Artifacts
Lawrance et al. (2008)	✓	Field notes	1-program comprehension	Qualitative
Piorkowski et al. (2011)	✓	Audio Think-aloud	1-program comprehension	Actions
Fritz et al. (2014)	✓	Patches	1-program comprehension	Actions
Duala-Ekoko and Robillard (2012)	✓	N.A.	1-unfamiliar APIs	Actions
Dekel and Herbsleb (2009)	✓	N.A.	1-unfamiliar APIs	Actions
Ko and Myers (2005)	✓	Audio	2-Tool design	Qualitative
Ko et al. (2005a)	✓	N.A.	2-Tool design	Actions
Ko et al. (2005b)	✓	N.A.	2-Tool design	Actions

Table 1 (continued)

Paper	Screen-captured Video[1]	Other Collection Method[2]	Research Purpose [3]	Video Coding Level
Hundhausen et al. (2006)	✓	N.A.	3-novice programming	Qualitative
Koru et al. (2005)	×	Video tape Think-aloud	3-pair programming	Qualitative
Dewan et al. (2009)	✓	Audio	3-distributed programming	Qualitative
Greiler et al. (2012)	×	Survey Interview	3-testing of plugin systems	Qualitative
Murphy-Hill et al. (2014)	×	Survey Interview	3-game development	Qualitative
Brade et al. (1992)	×	Video tape Think-aloud	4-code structure	Qualitative
Ammar and Abi-Antoun (2012)	✓	Think-aloud	4-code structure	Actions
Lawrence et al. (2005)	✓	Field notes Questionnaire	4-program execution	Qualitative
Sarma et al. (2009)	✓	Think-aloud	4-social relationships	Qualitative

[1] ✓: used, ×: not used

[2] N.A. means no other methods are used

[3] Research purpose category and specific aspect

Some of the surveyed papers also report the ratio of video recording time and coding time. The reported ratio is between 1:4–7, depending on the details and granularity of the HCI data to be collected. The most costly studies are to study fine-grained behavioral patterns in software development tasks (e.g., (Wang et al. 2011; Ko and Myers 2005)) because they require iterative open coding of screen-captured videos. For example, Ko and Myers (2005) reported "analysis of video data by repeated rewinding and fast-forwarding". However, compared to qualitative data collection and analysis methods, 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 Many studies on human aspects of software engineering have demonstrated the usefulness of screen-captured videos in studying developers' behavior in software development tasks. 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. We further elaborate on the kind of priori studies that could benefit from such an automated tool in Section 6.

3 Formative Study

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

3.1 Study Design

We recruit 3 graduate students from the School of Computer Engineering, Nanyang Technology University and ask them to manually transcribe a 20-minutes screen-captured task video. The 20-minutes task video is excerpted from the 29-hours of task videos in our previous field study of the developers' online search behavior in software development tasks (Li et al. 2013). The developers in that study use the Eclipse IDE to complete two programming tasks. They use Chrome, Internet Explorer, and Firefox to search the Internet and browse web pages. Details of this previous field study can be found in Section 5.1.

In this formative study, we instruct the three participants to extract the following three pieces of information from the task video:

1. The applications that the developer uses
2. The time and duration of application usage
3. The application content that the developer interacts with, including source files viewed, web pages visited, and search queries issued.

We instruct the three participants to log their manual transcription results in a table as shown in Table 2. A record in the table includes 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 starting time of two consecutive records. The participants are also asked to identify application usage with unique content and assign it a unique *Index*. The records in Table 2 show that the developer searches Google in Chrome and then visits a web page at help.eclipse.org. Next, he switches to

Table 2 An example of manual transcription logs

Index	Time	Application	Application Content
Url1	00:00	Chrome	URL: www.google.com Query:IProgressMonitor 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/...

Eclipse IDE and accesses two source files MyEditor.java and SampleAction.java. After that, he switches back to the eclipse help web page in Chrome. In this 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. Hence, they are assigned the same Index Url2.

Before the participants start coding the 20-minutes task video, we use a different 1-minute video and the manual transcription results to demonstrate to the three participants what information they need to extract and how to log their transcription results. The participants conduct a trial on this 1-minute video before they start their coding task.

3.2 Results

Table 3 shows how much time each participant spends in transcribing the 20-minutes task video and how many records he/she logs. 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 log is very different. The participant *S1* logs about 3 times more records than the participant *S3* does. We look into the transcription results of the three participants. We find that the participant *S1* considers screenshots with different content resulting from window scrolling in the same web page or source file as different application content, while the participant *S2* and *S3* do not consider so. This results in much more records in *S1*'s transcription results than that of *S2* and *S3*. Furthermore, the participant *S3* omits some application switchings whose duration are very short (i.e., switching from one application to another and then quickly switching back). This results in fewer records in *S3*'s transcription results than that of *S1* and *S2*.

In the 20-minutes screen-captured video, the developer uses two web search engines (Baidu and Google) and visits nine web pages. We further compare the search engines and web pages that the three participants log. Table 4 shows the results. Note that different web pages from the same website are annotated with an index number, such as topic.csdn.net (1), topic.csdn.net (2). We can see that the participant *S1* logs all the two search engines and the nine web pages, but both *S2* and *S3* miss three web pages. *S2* fails to recognize three

Table 3 The statistics of manual transcription by the three participants

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

Table 4 The transcription results of search engines or web pages visited

		S1	S2	S3
Search Engine	www.baidu.com	✓	✓	✓
	www.google.com.hk	✓	✓	✓
Visited Web Page	topic.csdn.net (1)	✓	✓	✓
	topic.csdn.net (2)	✓		✓
	hongyegu.iteye.com &	✓	✓	✓
	www.itpub.net	✓	✓	
	www.blogjava.net	✓	✓	✓
	docs.oracle.com	✓	✓	✓
	help.eclipse.org (1)	✓	✓	✓
	help.eclipse.org (2)	✓		
	help.eclipse.org (3)	✓		

different web pages from the same website, while *S3* fails to recognize two different web pages from the same website. *S3* fails to recognize www.itpub.net because the developer visits this web page very briefly.

Finally, we abstract each record in the transcription results of the three participants into a universal identifier (UID) based on the application content of the record, i.e., the query used, the web page visited, and the source file viewed in the record. As such, we obtain a sequence of UIDs for each participant. Figure 1 shows an illustrative example. Although we instruct the participants to log only web page visited and source file viewed, the participant *S1* interprets our instruction of unique content in a different way. He logs the same web page or source file with different content resulting from window scrolling (e.g., page up/down) as different records. As such, the sequence of UIDs of *S1*'s transcription results contains subsequences of the same UID. We replace the subsequence of the same UID with that UID. For example, a sequence of UIDs $\{Url1, Url2, Src1, Src1, Src2, Url2, Url3, Url3\}$ will be replaced as the sequence $\{Url1, Url2, Src1, Src2, Url2, Url3\}$. We then compute the Longest Common Subsequence (LCS) of the resulting sequence of UIDs of the two participants (S_i and S_j). We measure 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 5, the transcription results of different pairs of participants overlap to certain extent, but their transcription results are not very similar.

Summary This 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 constant attention to micro-level details. Otherwise, one is very like to miss some information. Finally, the transcription results by different participants can often be inconsistent.

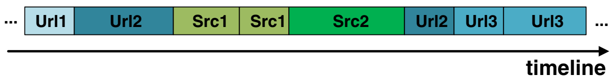


Fig. 1 An example of transcript

Table 5 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

Threats to Validity First, the study involves only three participants, and the task video used in the study is only 20 minutes. The limited transcription data may affect the analysis result. Second, this formative study asks the participants to watch the video and logs only their direct observation of application usage and application content, which can be considered as the lowest level of video analysis. The participants do not need to abstract developers' intentions and strategies.

However, one of the participants misinterprets our instruction about unique content, which results certain inconsistencies in the transcription results. This misinterpretation is not observed in the trial run of 1-minute video before the real task. We merge the resulting separate records in the transcription log of this participant to make his transcription log comparable with that of the other two participants. In addition to this transcription inconsistency, the inconsistencies between the participants' transcription results are mainly due to the error-prone nature of human attention and observation, such as ignoring briefly visited web pages, ignoring fast switching redundant between applications.

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 an 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 the definition of application windows as input to recognize screenshots in a screen-captured video. The definition of an application window "informs" the *scvRipper* tool with the window layout, the sample images of distinct visual cues of the window's GUI components, and the GUI components to be scraped once they are recognized.

Figure 2 shows the metamodel of application windows. *scvRipper* assumes that an application window is composed of a hierarchy of GUIComponents. Rows and windows define the layout of the application window. A row or window can contain nested rows, nested windows, and/or leaf GUIItems. Rows and GUIItems have relative positions in the application window (denoted as $index$), while windows do not. A GUIItem contains an ordered 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 $isunique$ field of the VisualCue is $true$. The GUIComponents whose $tobescraped$ field is true will be scraped from the application window in the screen-captured video.

Figure 3 shows the definition of the Eclipse IDE and the Google Chrome window. The definition of the Eclipse window assumes that the Eclipse window consists of a GUIItem (TitleBar) and four rows (Menu, ToolBar, MainContent, and StatusBar) from top down. We omit the definition details of Menu, ToolBar and StatusBar due to space limitation. The TitleBar contains a unique VisualCue (Eclipse application icon). MainContent row may contain CodeEditor windows and ConsoleView windows. CodeEditor window contains FileTab and EditArea GUIItems. These two GUIItems contain non-unique visual cues (such as Java file icons, compile error icons). This definition instructs *scvRipper* to scrape CodeEditor and ConsoleView windows from the Eclipse window.

The definition of the Chrome window assumes that the Chrome window consists of two rows from top down: Header and WebPage. The Header contains three GUIItems from left to right: NavigationPart, AddressBar, and Tool. NavigationPart contains three VisualCues from left to right: GoBack, GoForward, and Refresh buttons. These buttons are unique in the Chrome window. The WegPage may contain a SearchBox GUIItem as commonly seen in search engine web pages. 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 4 shows the screenshot of using configuration tool to define the Eclipse IDE window and the Google Chrome window shown in Fig. 3. Using the configuration tool, the user can intuitively define the application windows based on the GUI structure of the application windows. Collecting sample images of visual cues may require certain efforts. However, this task usually needs to be done only once. As long as applications use the same layout and window structure, the definition of an application window can be reused for screen-captured videos taken on different computers with 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.

4.2 Technique Overview

Figure 5 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.

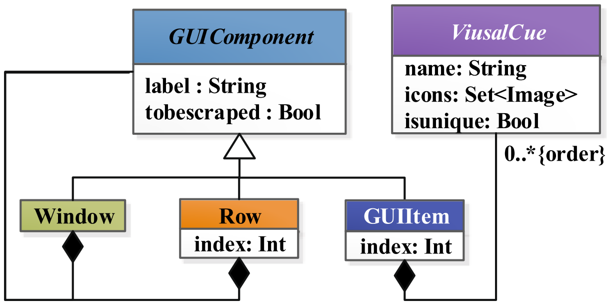
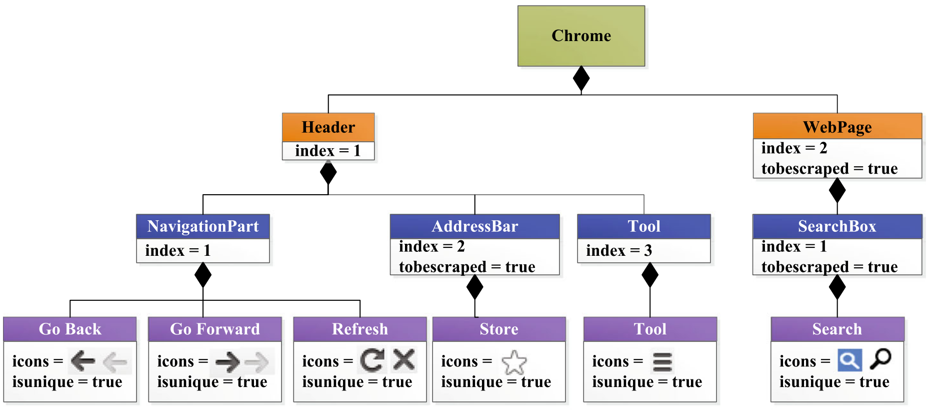
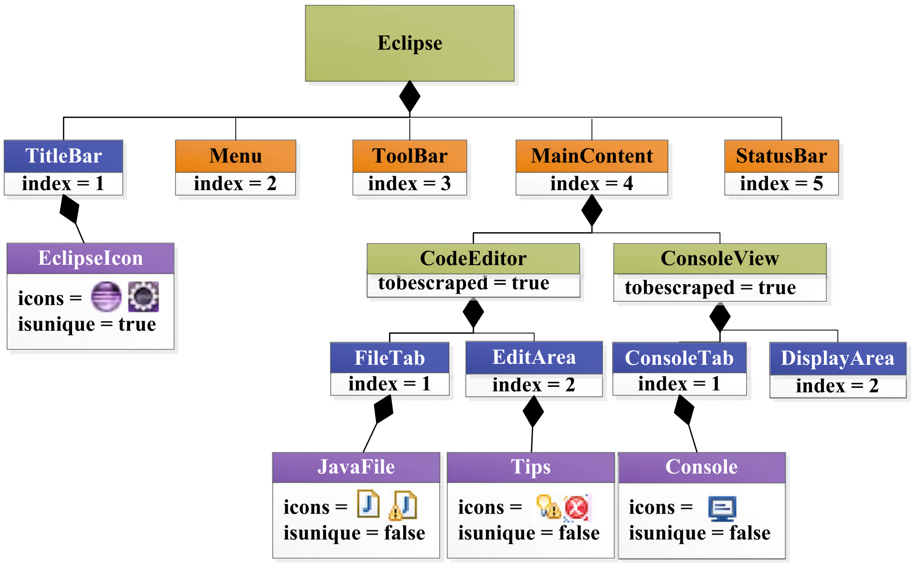


Fig. 2 The metamodel of application windows



(a) Definition of Google Chrome Window

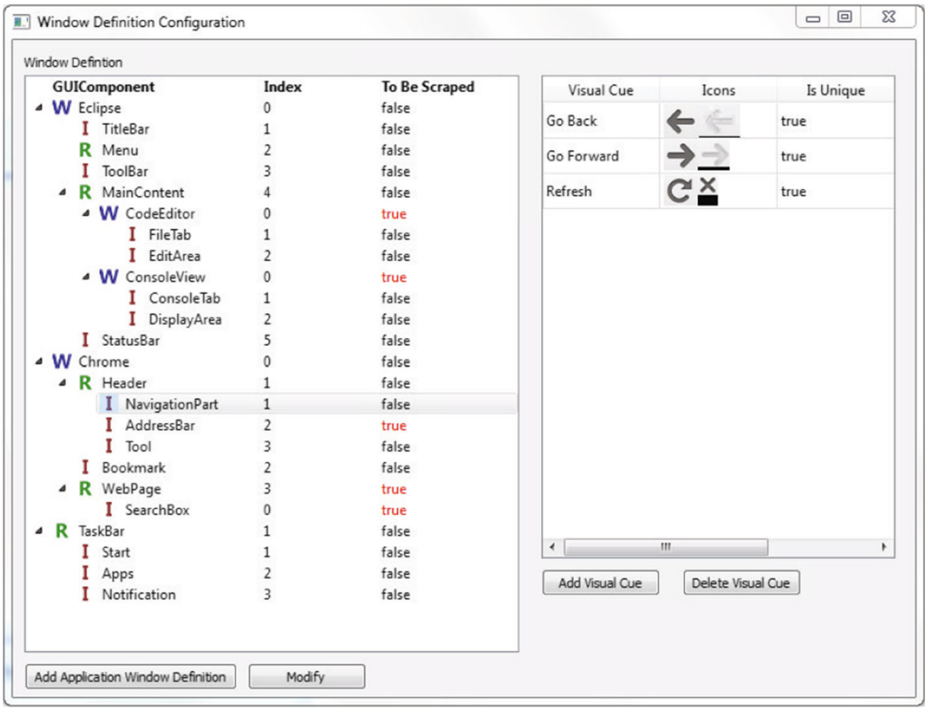


(b) Definition of Eclipse IDE Window

Fig. 3 Two instances of application-window metamodel

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 a human can interpret into a time-series HCI data that a computer can automatically analyze.

First, *scvRipper* uses image differencing technique (Wu and Tsai 2000) to detect screen-shots 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 screenshot at a



Window Definition Configuration

Window Defintion

GUIComponent	Index	To Be Scraped
W Eclipse	0	false
I TitleBar	1	false
R Menu	2	false
I ToolBar	3	false
R MainContent	4	false
W CodeEditor	0	true
I FileTab	1	false
I EditArea	2	false
W ConsoleView	0	true
I ConsoleTab	1	false
I DisplayArea	2	false
I StatusBar	5	false
W Chrome	0	false
R Header	1	false
I NavigationPart	1	false
I AddressBar	2	true
I Tool	3	false
I Bookmark	2	false
R WebPage	3	true
I SearchBox	0	true
R TaskBar	1	false
I Start	1	false
I Apps	2	false
I Notification	3	false

Visual Cue	Icons	Is Unique
Go Back	←	true
Go Forward	→	true
Refresh	↻ ✕	true

Fig. 4 The configuration tool for window definition

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 Fig. 6 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

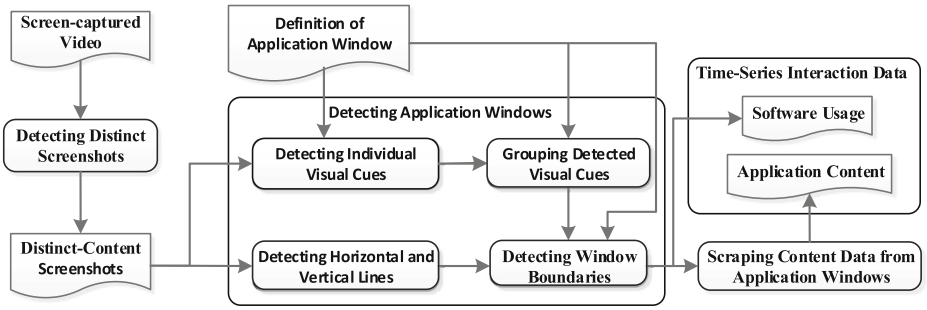

Screen-captured Video
Definition of Application Window
Detecting Distinct Screenshots
Detecting Application Windows
Detecting Individual Visual Cues
Grouping Detected Visual Cues
Time-Series Interaction Data
Software Usage
Application Content
Distinct-Content Screenshots
Detecting Horizontal and Vertical Lines
Detecting Window Boundaries
Scraping Content Data from Application Windows

Fig. 5 The process of our video scraping technique to extract time-series HCI data

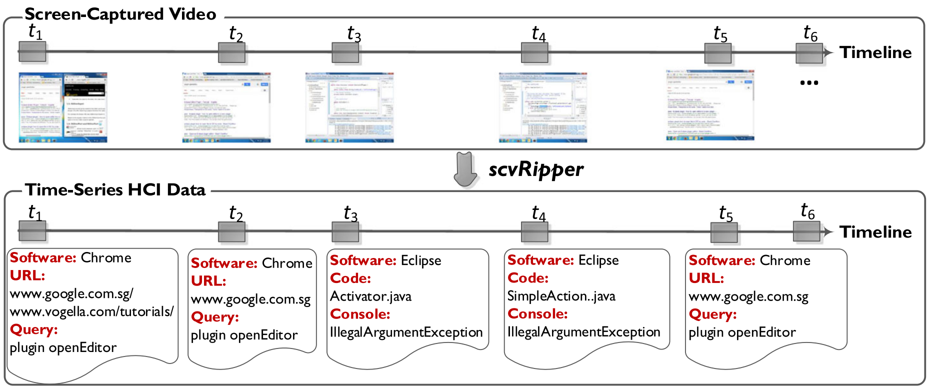


Fig. 6 An illustrative example of a screen-captured video and video scraping results

of Fig. 6 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 Fig. 3. 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 uses at different time periods. It also identifies the application content that the developer accesses and/or generates (such as search queries, web pages, code fragments, and console output) at different time periods. Further information (e.g., method name, exception) can be extracted from the application content through post analysis of video scraping data.

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, for example, mouse or cursor movement, small window movement, menu display, several lines of scrolling. This produces a sequence of distinct consecutive screenshots, s_1, s_2, ..., s_n where any two consecutive screenshots s_i and s_{i+1} are different, i.e., over a user-specified threshold (t_{diff}). The two non-consecutive screenshots can still be the same in this sequence of distinct consecutive screenshots. *scvRipper* uses image differencing technique again to identify distinct-content screenshots. *scvRipper* stores the traceability between a distinct-content screenshot and all the screenshots it represents during this image differencing process.

Take the screen-captured video in Fig. 6 as an example. The developer views two web pages side-by-side in the two Chrome windows. He then maximizes one of the Chrome windows. After a while, he switches from the Chrome window to an Eclipse IDE window. He opens two different methods in Eclipse and read the code. Next he switches from the Eclipse window back to the Chrome window. Assume this sequence of human-computer interaction takes 120 seconds. A 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 Fig. 6. It can then identify that the screenshots at timestamp t_2 and t_5 are the same. The screenshots at timestamp t_1 and t_2 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

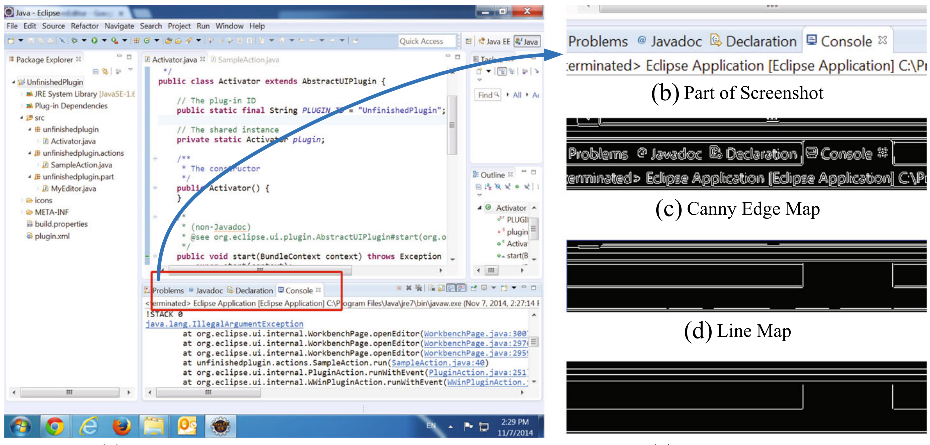
The core algorithm of *scvRipper* takes as input a distinct-content screenshot and the definition of application windows to be recognized in the screenshot. It recognizes application windows in the screenshot in four steps:

1. Detecting horizontal and vertical lines.
2. Detecting individual visual cues.
3. Grouping detected visual cues.
4. Detecting window boundaries.

scvRipper can accurately recognize stacked or side-by-side windows.

4.4.1 Detecting Horizontal and Vertical Lines

Figure 7 illustrates the process of detecting horizontal and vertical lines. Figure 7a is the screenshot of the Eclipse window at time period $t_3 - t_4$ in Fig. 6. *scvRipper* assumes that an application window (or subwiondow) has explicit window boundaries and occupies a rectangular region in the screenshot. Thus, *scvRipper* first uses the canny edge detector (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 7c shows the canny edge map of the part of the screenshot in Fig. 7b.



```
 */
public class Activator extends AbstractUIPlugin {

    // The plug-in ID
    public static final String PLUGIN_ID = "UnfinishedPlugin";

    // The shared instance
    private static Activator plugin;

    /**
     * The constructor
     */
    public Activator() {
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.ui.plugin.AbstractUIPlugin#start(org.o
     */
    public void start(BundleContext context) throws Exception
```

```
<terminated> Eclipse Application [Eclipse Application] C:\Program Files\Java\jre7\bin\javaw.exe (Nov 7, 2014, 2:27:14 I
!STACK 0
java.lang.IllegalArgumentException
        at org.eclipse.ui.internal.WorkbenchPage.openEditor(WorkbenchPage.java:300
        at org.eclipse.ui.internal.WorkbenchPage.openEditor(WorkbenchPage.java:297
        at org.eclipse.ui.internal.WorkbenchPage.openEditor(WorkbenchPage.java:295
        at unfinishedplugin.actions.SampleAction.run(SampleAction.java:40)
        at org.eclipse.ui.internal.PluginAction.runWithEvent(PluginAction.java:251
        at org.eclipse.ui.internal.WWinPluginAction.runWithEvent(WWinPluginAction.
```

Problems Javadoc Declaration Console
erminated> Eclipse Application [Eclipse Application] C:\Pr

(a) A Screenshot of Eclipse IDE Window

(b) Part of Screenshot

(c) Canny Edge Map

(d) Line Map

(e) Clustered Line Map

Fig. 7 An example of detecting horizontal and vertical lines

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 removes the horizontal lines whose length is less than K. For vertical lines, erosion followed by dilation with the kernel $[1]_{K\times 1}$ (i.e., a vertical line of K pixels) on the edge map removes the vertical lines whose length is less than K. These erosion and dilation operations generate a line map of the screenshot (see Fig. 7d).

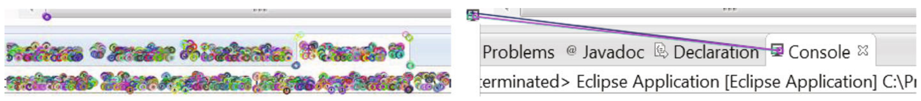
The horizontal (or vertical) lines in the line map can be very close to each other. Such close-by horizontal (or vertical) lines introduce noises and increase complexity to detect the window boundaries. Given a line map of the screenshot, *scvRipper* uses density-based clustering algorithm (DBSCAN Ester 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 8a visualizes the key points image of the part of the screenshot in Fig. 7b. The left corner of Fig. 8b visualizes the key points image of the visual cue of ConsoleView of Eclipse window. *scvRipper* detects the occurrence of this visual cue in the screenshot as indicated by the lines in Fig. 8b.

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



erminated> Eclipse Application [Eclipse Application] C:\P

(a) Key Points of Part of Screenshot in Fig 7(b) (b) Key Point Matching

Fig. 8 An example of detecting individual visual cues

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 the similarity 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

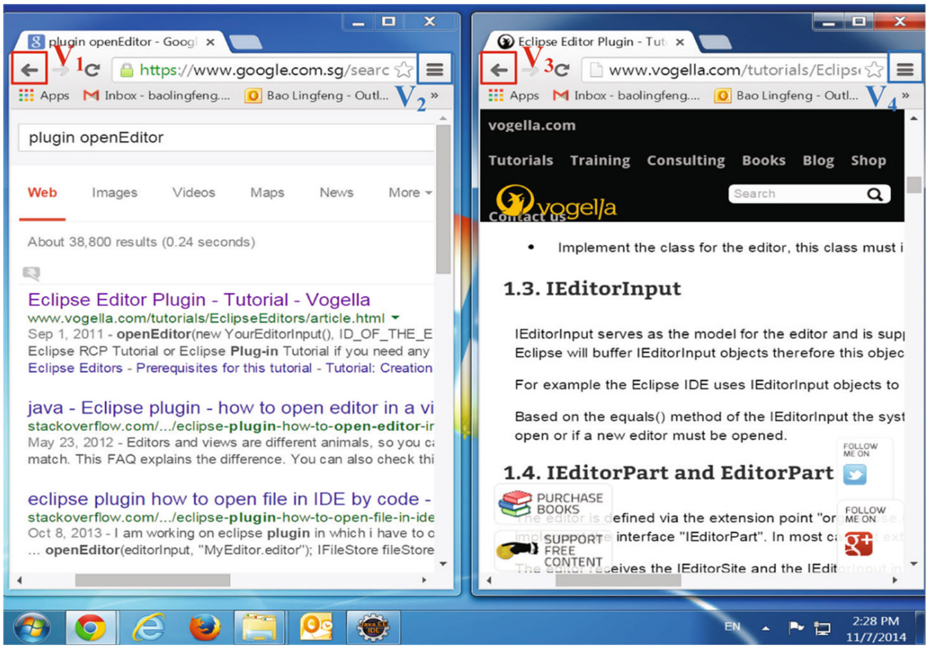
A screenshot may or may not contain the application windows of interest. To determine if the screenshot contains the window(s) of a given application, *scvRipper* counts the number of the detected visual cues that belong to the application according to the definition of the application window. Multiple 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* sets their affinity to 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 to 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 9 shows the screenshot of the two side-by-side Chrome windows at time period $t_1 - t_2$ in Fig. 6. 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.



plugin openEditor

About 38,800 results (0.24 seconds)

Eclipse Editor Plugin - Tutorial - Vogella
www.vogella.com/tutorials/EclipseEditors/article.html
Sep 1, 2011 - **openEditor**(new YourEditorInput(), ID_OF_THE_E
Eclipse RCP Tutorial or Eclipse **Plug-in** Tutorial if you need any
Eclipse Editors - Prerequisites for this tutorial - Tutorial: Creation

java - Eclipse plugin - how to open editor in a vi
stackoverflow.com/.../eclipse-**plugin-how-to-open-editor**-ir
May 23, 2012 - Editors and views are different animals, so you ca
match. This FAQ explains the difference. You can also check thi

eclipse plugin how to open file in IDE by code -
stackoverflow.com/.../eclipse-**plugin**-how-to-open-file-in-ide
Oct 8, 2013 - I am working on eclipse **plugin** in which i have to o
... **openEditor**(editorInput, "MyEditor.editor"); IFileStore fileStore

- Implement the class for the editor, this class must i

1.3. IEditorInput

IEditorInput serves as the model for the editor and is sup
Eclipse will buffer IEditorInput objects therefore this objec

For example the Eclipse IDE uses IEditorInput objects to

Based on the equals() method of the IEditorInput the syst
open or if a new editor must be opened.

1.4. IEditorPart and EditorPart

The editor is defined via the extension point "or
implementation interface "IEditorPart". In most ca
The editor receives the IEditorSite and the IEdit

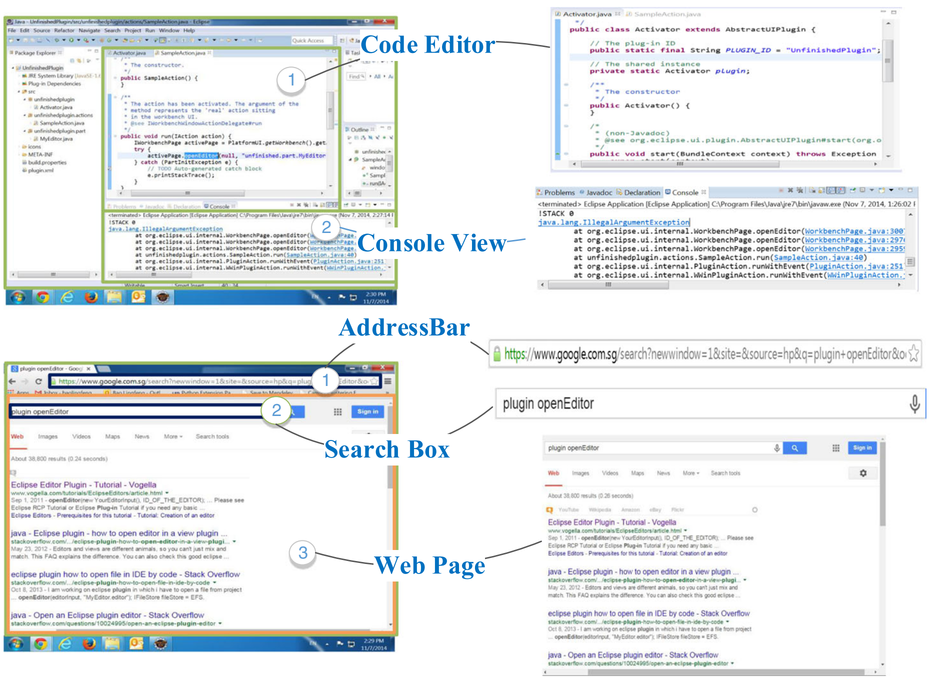
Fig. 9 An example of affinity calculation

Given the two visual cues whose affinity is not yet set to 0 based on the uniqueness and relative positions of the visual cues, *scvRipper* computes their affinity as $e^{-(d_{ij}^2/\delta^2)}$ where d_{ij} is the distance between the center of the two visual cues V_i and V_j and δ is a term proportional to the image size. Intuitively, the more the distance between the two visual cues, the less likely the two visual cues belong to the same application window. In Fig. 9 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 10 shows the detected boundaries of the Eclipse window (at time period $t_3 - t_4$ in Fig. 6) and the Chrome window (at time periods $t_2 - t_3$ and $t_5 - t_6$ in Fig. 6). It also shows the detected boundaries of the to-be-scraped GUIComponents in the two windows.



```
 */
public class Activator extends AbstractUIPlugin {

    // The plug-in ID
    public static final String PLUGIN_ID = "UnfinishedPlugin";

    // The shared instance
    private static Activator plugin;

    /**
     * The constructor
     */
    public Activator() {
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.ui.plugin.AbstractUIPlugin#start(org.o
     */
    public void start(BundleContext context) throws Exception
```

```
<terminated> Eclipse Application [Eclipse Application] C:\Program Files\Java\jre7\bin\javaw.exe (Nov 7, 2014, 1:26:02 F
!STACK 0
java.lang.IllegalArgumentException
        at org.eclipse.ui.internal.WorkbenchPage.openEditor(WorkbenchPage.java:300
        at org.eclipse.ui.internal.WorkbenchPage.openEditor(WorkbenchPage.java:297
        at org.eclipse.ui.internal.WorkbenchPage.openEditor(WorkbenchPage.java:295
        at unfinishedplugin.actions.SampleAction.run(SampleAction.java:40)
        at org.eclipse.ui.internal.PluginAction.runWithEvent(PluginAction.java:251
        at org.eclipse.ui.internal.WWinPluginAction.runWithEvent(WWinPluginAction.
```

https://www.google.com.sg/search?newwindow=1&site=&source=hp&q=plugin+openEditor&o

plugin openEditor

About 38,800 results (0.26 seconds)

Eclipse Editor Plugin - Tutorial - Vogella
www.vogella.com/tutorials/EclipseEditors/article.html
Sep 1, 2011 - openEditor(new YourEditorInput(), ID_OF_THE_EDITOR); ... Please see Eclipse RCP Tutorial or Eclipse Plug-in Tutorial if you need any basic ...
Eclipse Editors - Prerequisites for this tutorial - Tutorial: Creation of an editor

java - Eclipse plugin - how to open editor in a view plugin ...
stackoverflow.com/.../eclipse-plugin-how-to-open-editor-in-a-view-plugi...
May 23, 2012 - Editors and views are different animals, so you can't just mix and match. This FAQ explains the difference. You can also check this good eclipse ...

eclipse plugin how to open file in IDE by code - Stack Overflow
stackoverflow.com/.../eclipse-plugin-how-to-open-file-in-ide-by-code
Oct 8, 2013 - I am working on eclipse plugin in which i have to open a file from project ... openEditor(editorInput, "MyEditor.editor"); IFileStore fileStore = EFS.

java - Open an Eclipse plugin editor - Stack Overflow
stackoverflow.com/questions/10024995/open-an-eclipse-plugin-editor

Fig. 10 An example of boundary detection and image scraping results

The detected boundaries are highlighted in the same color as that of the corresponding type of GUIComponent in Fig. 2.

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 10 presents an example of the image scraping results of the Eclipse window and the Chrome window. The textual data from OCR records the contents that 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 use the 29-hours of screen-captured task videos from our previous study (Li et al. 2013) to evaluate the runtime performance and effectiveness of our *scvRipper* tool. To evaluate the tool's data extraction capability and the usefulness of the extract HCI data, we analyze the developers' online search behavior in software development using the time-series HCI data extracted from the 29 h of task videos. The analysis of the developers' micro-level behavior patterns enabled by the *scvRipper* tool leads to several implications for enhanced tool supports for online search in software development.

5.1 Data Set

The data we use is the screen-captured videos in our previous study of the developers' online search behavior during software development (Li et al. 2013). We use these screen-captured videos because of the diversity of the development tasks involved in our previous study and the diversity of the computer settings and software tools that the participants use. These diversities make the videos a good test bed for evaluating the runtime performance and robustness of our *scvRipper* tool and usefulness of the video scraping data. Note that our previous study performs only manual, qualitative analysis of the task videos. As such, it investigates only the general information needs and search processes in software development tasks. As shown in this section, the *scvRipper* tool can extract much more fine-grained HCI data from the task videos to study the developers' micro-level behavior patterns.

Our previous study includes two software development tasks. The first task (Task1) is to develop a new P2P chat software. 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. 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).

Eleven graduate students are recruited in the first task, and 13 different graduate students are recruited in the second task from the School of Computer Science, Fudan University. The participants have general Java programming experience. As our goal in the previous study is to investigate the developers' online search behavior in software development tasks, we select the participants who do not possess all the knowledge necessary for the tasks, for example, Java multi-threading or socket APIs for the first task, and Eclipse plugin development or Eclipse editor API for the second task. Thus, the participants have to interleave coding, web search and learning during the tasks.

The participants are instructed to use a screen-capture software to record their working process. They use their own computers that have different window resolutions and color schema. As the task videos of 4 participants are corrupted, we use the 29 h task videos of the 20 participants (8 from the first task and 12 from the second task) to evaluate our video scraping tool. Figure 11 shows the box-plot of the Task Video Length (TVL in minutes) of these participants.

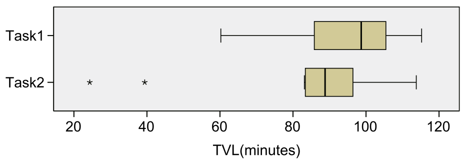


Fig. 11 The statistics of Task Video Length (TVL)

5.2 Data Extraction and Processing

Based on the software tools that the participants used in our previous study, we define application windows for the *scvRipper* tool to recognize Eclipse IDE window and web browser window (Google Chrome, Mozilla Firefox, Internet Explorer). Figure 3 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. 10 for an example).

Given a task video, *scvRipper* automatically extracts time-series HCI data from screen-captured videos, including the applications being used and the application content as defined above. Examples of the extracted time-series HCI data are illustrated in Figs. 6 and 10. We then post-process the extracted time-series HCI data to compute and analyze various statistics of application usage (e.g., within-application switching, between-application switching) and content usage (e.g., most visited websites, keyword source, query refinement). These application-usage and content-usage statistics can be easily obtained using tools like Matlab. However, the most challenging part of this case study is to obtain the HCI data that can be programmatically analyzed from the video that can only be interpreted by human.

5.3 Runtime Performance

We run our *scvRipper* tool on one Windows 7 computer with 4GB RAM and Intel(R) Core(TM)2 Duo CPU. The 29 h task videos are recorded at sample rate 5 screenshots per second. As such, the 29 h task videos consist of in total over 520K screenshots. Our *scvRipper* tool takes 43 h to identify about 11K distinct-content screenshots from the 29 h 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 takes about 122 h to extract time-series HCI data from the 11K distinct-content screenshots, i.e., on average 38.41 ± 16.94 seconds to analyze one distinct-content screenshot. The OCR of the scraped image content takes about 60 h .

The current implementation of *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[2] could also reduce the time of detecting distinct-content screenshots and the OCR of scraped screen images.

5.4 Robustness

We qualitatively evaluate the robustness of the key steps of the *scvRipper* tool (i.e., image differencing algorithm, application window detection, and OCR of scraped screen images)

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

using randomly sampled 500 distinct-content screenshots from different developers' task videos at different time periods.

First, we examine the video clips that these sampled distinct-content screenshots represent and find that the *scvRipper*'s image differencing algorithm (at $t_{diff} = 0.7$ in this study) can tolerate the reasonable differences between the screenshots caused by window scrolling, mouse or cursor movement, and menu display, and minor code editing. Ignoring these screenshots should not cause significant information loss for data analysis.

Second, we examine 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 are 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 a failure in 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 the 500 sampled screenshots (less than 1 %).

Third, we evaluate 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 have 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 use scales the low DIP screenshots to 300 DPI and produces acceptable OCR results.

We also compare the HCI data extracted by the *scvRipper* tool from the 20 min task video with the manual transcription results of the three human participants on the 20 min task video obtained in the formative study. The *scvRipper* tool recognizes 277 distinct screenshots from the 20 min video, which is more than the number of records transcribed by the human participants in the formative study (See Table 3 in Section 3). This is because there are lots of small changes between adjacent screenshots which are usually ignored by the human participants, e.g. the change caused by window scroll up/down. The *scvRipper* tool identifies 8 distinct web pages from the 20 min video but misses the web page "help.eclipse.org (3)" (See Table 4). This is because this web page is very similar to anther web page on the "help.eclipse.org" web site and the duration on this web page is very short. Thus, the *scvRipper* tool misses the web page.

5.5 Data Analysis on Online Search

Next, we describe examples of fine-grained HCI data that the *scvRipper* tool can extract from the task videos and the usefulness of the extracted data for understanding the developers' online search behavior patterns in software development.

5.5.1 Most Visited Web Sites

First, we extract web site names (i.e., domain name) from the scrapped URLs. We categorize the web sites that the developers visited during the two tasks into seven 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 6 lists the top

Table 6 The top three most-visited web sites of 7 web categories

	The Top 3 Most-Visited Web Sites	#Frequency
Search engines (SE)	www.baidu.com	121
	www.google.com	62
	www.bing.com	8
Doc sharing sites (DS)	www.360doc.com	21
	www.doc88.com	8
	www.docin.com	6
Technical tutorials(TT)	blog.csdn.net	28
	developer.51cto.com	12
	www.newasp.cn	9
Topic forums(TF)	topic.csdn.net	11
	www.newsmth.net	6
	java.chinaitlab.com	5
Code hosting sites (CH)	download.csdn.net	13
	code.google.com	5
	github.com	3
Q&A websites (QA)	zhidao.baidu.com	10
	iask.sina.com.cn	4
	stackoverflow.com	2
API specification (API)	docs.oracle.com	7
	developers.google.com	5
	www.aspose.com	3

three most visited web sites of these seven categories in our study. The results show that developers heavily relied on search engines, doc sharing sites and technical tutorials in the two development tasks.

5.5.2 *Web Page Visited After a Search*

We consider the developer visiting a web page if the *scvRipper* tool extracts the URL of the web page from the task videos, and the developer performing a search if the *scvRipper* tool extracts the search engine web page with a query. Figure 12 presents the times that the developers visited a specific number of unique URLs (i.e., web pages) after a search in the two tasks. We can see that the developers in the first task open much less number of web pages after a search than the developer in the second task does. This reflects the complexity of the two tasks and the information needs of the developers in the two tasks. We further elaborate on this in Section 5.5.4.

5.5.3 *Web Page Visited and Web Page Switching During the Tasks*

Figure 13 shows the number of unique URLs (i.e., web pages) that the 20 developers visit in the two tasks and the number of switchings between these web pages. In the first task, 5 developers visit less than 9 web pages and make less than 9 web-page switchings. However,

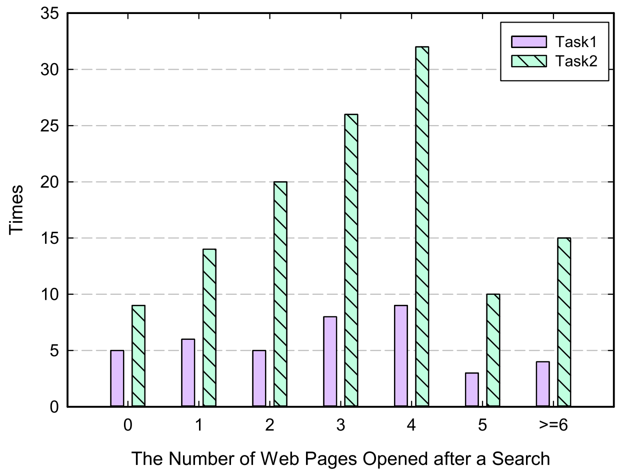


Fig. 12 The statistics of opening a specific number of web pages

the other 3 developers visit on average 20 ± 4 web pages and make on average 37.6 ± 19.5 web-page switchings. In the second task only 2 developers visit less than 6 web pages and make less than 5 web-page switchings. These two developers are experienced Eclipse plugin developers. They complete the task much faster than the other 10 less experienced developers (i.e., the two outliers '*' in Fig. 11). During the task, they only issue a few searches and explore a small number of web pages. The other 10 developers visit on average 31.4 ± 13.1 web pages and make on average 75.7 ± 38.1 web-page switchings.

The results show that in less than two hours of tasks most of the developers have to explore, examine, and use a large amount of information when searching and using online resources. We observe two common reasons why developers sometimes visit a large number of web pages and switched between them. First, the developers need to select the most relevant web pages from several web pages, or to integrate information from several web pages. For example, the developers D2 issues only 2 new queries. But he prefers to visit

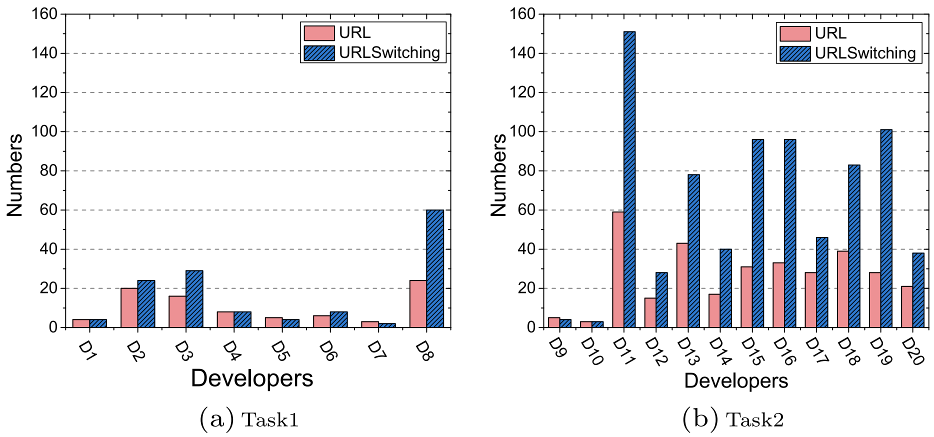


Fig. 13 Statistics of unique URLs and URL switchings

several web pages from the search results, and then examines these web pages to determine their relevance. He visits in total 20 web pages and makes 24 web-page switchings.

Second, the developers need to follow information scents from one web page to another during their online search process. For example, the developer D15 searches “PermGen space”. He opens a web page about “PermGen space error” on “http://www.cnblogs.com” from the search results. He follows the link on this web page to another web page on “http://blog.csdn.net” about “JVM parameter setting”. This csdn web page helps him fix the virtual memory issue to run the Eclipse IDE.

5.5.4 *Keyword Source in Queries*

Given a search query extracted from search results web pages, we determine the sources of its keywords by searching code fragments and console output extracted from the distinct Eclipse IDE screenshots before the web-page screenshot in which a keyword is used for the first time. If a keyword appears in code fragments, for example, the keyword “openEditor” in the query “java.lanq.IllegalArgumentException openEditor” is an Eclipse API used in the source code, we consider 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 consider its source as “FromConsole”. If a keyword appears in both code fragments and console outputs, we consider the keyword as “FromCode”. If a keyword appears in neither code fragments nor console outputs, we consider its source as self-phrasing, for example, the keywords “eclipse” and “rcp” in the query “eclipse rcp EditorPart EdtorInput”.

Table 7 summarizes the number of distinct keywords that the developers use in the two tasks and the sources of these keywords. In the first task the developers' keywords are mainly self-phrased. In the second task the keywords are both self-phrased and from IDE context.

Table 8 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 are considered as self-phrasing. Three of these 7 keywords are 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 search 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 are considered as self-phrasing, three of which describe application platform and techniques to be used (Eclipse, plugin, SWT) and three are from task description (editor, view, savefile). The other 5 most-used keywords are from IDE context, which describe Eclipse APIs required for the task (*EditorPart*, *openEditor*, *IEditorInput*, *doSave*, *IWorkbenchPage*). In the second task, developers have to fix bugs in using specific Eclipse APIs and extend specific Eclipse interface. However, only using specific Eclipse APIs often cannot find good online examples to accomplish the second task. Developers have to use application and task context to restrict the search.

5.5.5 *Query Refinement*

Given the search queries that a developer issued, we measure 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., $|Q_i \bigcap Q_{i+1}|/|Q_i \bigcup Q_{i+1}|$. If the Jaccard coefficient of the two consecutive

Table 7 Statistics of distinct keywords and keyword sources

Developer ID	#Query	#DistinctKW	#FromCode	#FromConsole	#Self-phrasing
Taks1					
D1	2	2	1	0	1
D2	3	5	0	0	5
D3	8	13	2	2	9
D4	7	12	0	0	12
D5	2	5	1	0	4
D6	5	13	0	0	13
D7	6	9	2	1	6
D8	8	13	1	0	12
mean±standard deviation(D1–D8)	5.13±2.37	9±4.15	0.87±0.78	0.37±0.69	7.75±4.11
Task2					
D9	4	10	2	0	8
D10	2	2	1	0	1
D11	22	32	14	6	12
D12	6	10	3	1	6
D13	7	9	1	0	8
D14	15	16	5	1	10
D15	9	13	1	5	7
D16	15	20	3	3	14
D17	12	18	5	0	13
D18	16	18	7	0	11
D19	7	9	5	0	4
D20	12	15	6		9
mean±standard deviation(D9-D20)	10.58±5.54	14.33±7.5	4.41±3.49	1.33±2.05	8.58±3.61

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.IllegalArgumentException" of the developer D11 is considered as a refinement of his 3rd query "java.lanq.IllegalArgumentException", while his 5th query "eclipse rcp EditorPart EdtorInput" is considered as a new query as it is very different from the 4th query.

Figure 14 shows the number of new queries and the number of query refinements in the two tasks. In the first task the developers issue in total 27 new queries (on average 3.37±1.34 new queries per developer). The developers do not refine 7 of these 27 new queries. They refine 12 new queries 1–3 times, and 2 new queries more than 3 times. The rest 6 queries are too different from their preceding queries, and thus are considered as new queries. In the second task the developers issue in total 57 new queries (on average 3.37 ± 2.18 new queries per developer). Nine of these 57 new queries are not refined. 30 new queries are refined 1–3 times, and 9 new queries are refined more than 3 times. The rest 4 queries are considered as new queries because they are too different from their preceding queries.

Table 8 Most-used keywords in the two tasks

Keywords	Frequency (times)	Who Used These Keywords
Task1		
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
Task2		
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

5.5.6 Search Frequencies and Intervals

The extracted time-series HCI data identifies the search queries that the participants issued through the tasks. We consider 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 collect the interval time of the two consecutive searches with different queries (denoted by τ) of

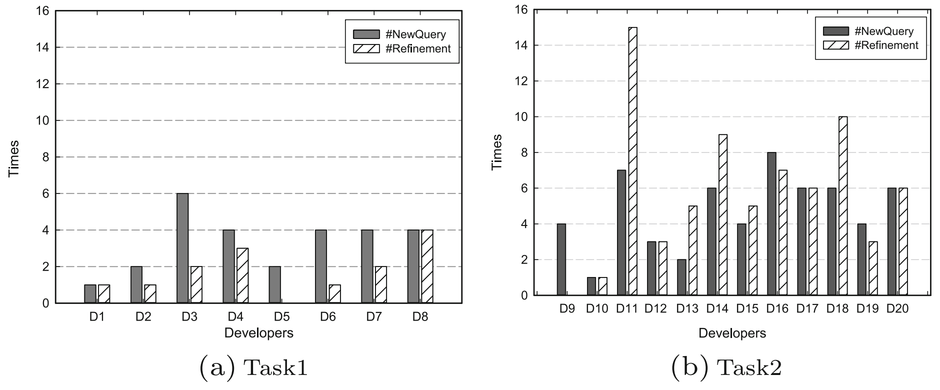


Fig. 14 The statistics of new queries and query refinements

the 20 developers in the two tasks. We use probability density function $p(\tau)$ to describe the relative likelihood of the interval time of two consecutive searches between a given interval. We obtain 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. 15.

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 λ is exponent parameter and k is a constant coefficient. We fit our data samples of interval time of two consecutive searches in terms of this equation using the Least Squares Fitting (Weisstein 2011). The fitting result is shown in red line in Fig. 15. This red line is $p(\tau) = \frac{1}{1.41} \times 0.45e^{-0.45\tau}$. We employ coefficients of determination R^2 (Colin Cameron and Windmeijer 1997) to determine how well our experimental data fit the statistical model. The R^2 is 0.97 which indicates that our data samples can be well explained by the statistical model represented by the red line.

Given the probability density function $p(\tau)$, the probability of variable τ ranging from τ_1 to τ_2 is equal to $P(\tau_1 < \tau \leq \tau_2) = \int_{\tau_1}^{\tau_2} p(\tau)d\tau$ (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 search with a different query within 1 minute is 0.48, within 3 min is 0.68, and within 10 min is 0.86.

5.6 Data Analysis on Context Switching

This section analyzes the developers' context switching activities within and across the IDE and web browser using the HCI data extracted by the *scvRipper* tool. 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

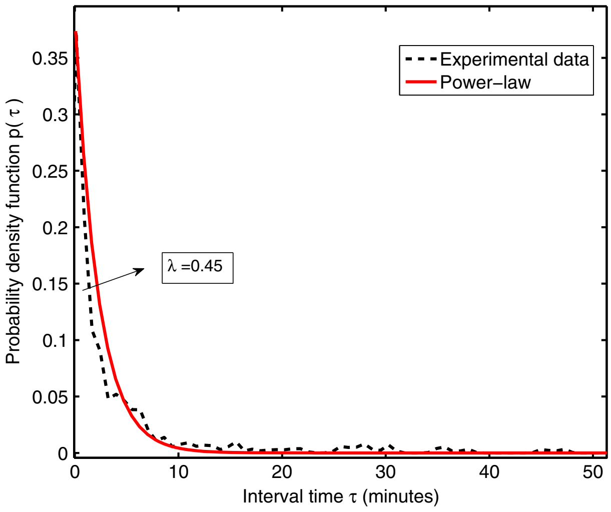


Fig. 15 The distribution of interval time of two consecutive queries

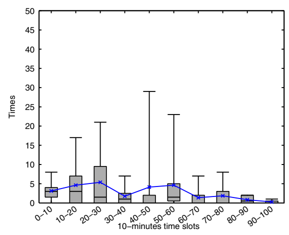
web content (Within-Browser switching). We also compute the time that the developers spend on the distinct IDE contents and the distinct web contents in the two tasks.

5.6.1 *Working Context Switching*

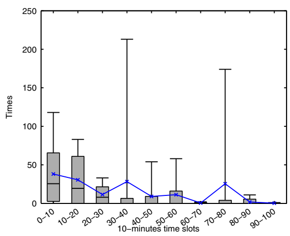
First, we want to compare how the developers progress through the two tasks. To make the developers' progress comparable, we split their task processes into 10-minutes buckets and aggregate their context switching activities in these 10-minutes buckets.

Figure 16 shows the number of IDE $\rightleftharpoons$ Browser switchings, Within-Browser switchings, and Within-IDE switchings that the developers perform in every 10-minutes bucket 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 perform in a 10-minute time slot. The band inside the box is the second quartile (Q_2, i.e., the median). The gray boxes indicate the interquartile range ($IQR = Q3 - Q1$). The lowest end of the whiskers represents minimal observation, and the highest end of whiskers represents maximal observation. The blue line shows the mean values of the number of switchings over time.

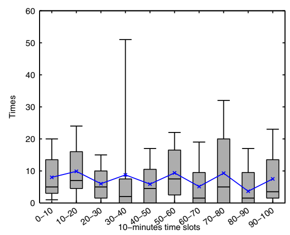
In the first task the developers start with a small number of IDE $\rightleftharpoons$ Browser switchings and a large number of Within-Browser switchings and Within-IDE switchings in the first 10 minutes. This indicates that the developers are trying to understand the problem they need to solve. The developers' Within-Browser and Within-IDE switchings remain relatively stable or drop in the next 20 min, while the IDE $\rightleftharpoons$ Browser switchings increase at the same time. This indicates that the developers find good online examples and start integrating online examples in the IDE. Then, the developers' Within-Browser and IDE $\rightleftharpoons$ Browser switchings drop for the rest of the first task, while the developers' Within-IDE switchings remain active. That is, the developers focus on developing the software within the IDE without much need for further online search.



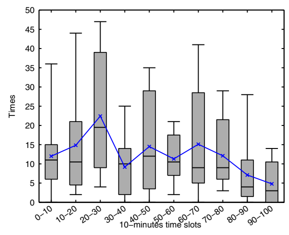
(a) IDE $\rightleftharpoons$ Browser in Task1



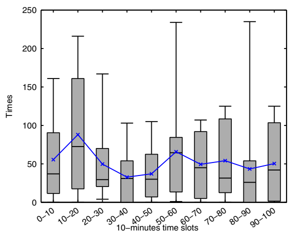
(b) Within-Browser in Task1



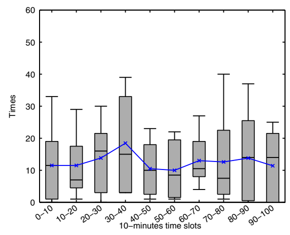
(c) Within-IDE in Task1



(d) IDE $\rightleftharpoons$ Browser in Task2



(e) Within-Browser in Task2



(f) Within-IDE in Task2

Fig. 16 Statistics of application and content switchings in every 10 minutes

In the second task the developers also start 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 min. Next, there is a surge in the Within-Browser switchings in the 11–20 min followed by a surge in the IDE $\rightleftharpoons$ Browser switchings in 20–30 min. Similar to the first task, the developers find some useful online resources and start integrating them into the IDE in the first 30 min. However, the Within-Browser and IDE $\rightleftharpoons$ Browser switchings are much more intense in the second task than in the first task. Furthermore, the Within-Browser and IDE $\rightleftharpoons$ Browser switchings do not drop after the 30 min in the second task. Unlike the first task in which the developers' search activity occur mainly in the beginning of the task, the developers in the second task have to frequently search and integrate online resources for the emerging problems throughout the task.

5.6.2 Markov Model on Context Switching

To further study the implicit information flow within web browser and between IDE and web browser, we build 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 6) 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 9 presents the transition probabilities of the Markov Model. The maximal probability of each row is highlighted in bold font.

The table shows that the developers have 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 task. There are 5 developers who download code examples from some technical tutorials and customize these code examples to complete the first task. In addition, the developer also integrate the information found on Q&A sites (QA) and API specification sites (API) into 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 have 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 exhibit 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 have the 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 are not the dominant information sources in the second task. The developers need more diverse information from different sources. Furthermore, the developers are more likely to integrate the information found on these information sources, instead of using the information to refine their search.

Table 9 Markov transition matrices

		Destination States							
		Eclipse	SE	TT	DS	TF	CH	QA	API
(a) Task 1									
Source states	Eclipse	0	0.16	**0.62**	0.03	0.11	0	0.07	0.01
	SE	0.27	0	**0.34**	0.08	0.20	0.05	0.06	0
	TT	**0.73**	0.20	0	0	0.06	0	0.01	0
	DS	0.38	**0.50**	0	0	0.13	0	0	0
	TF	0.38	**0.45**	0.07	0	0	0	0.07	0.03
	CH	0.33	**0.67**	0	0	0	0	0	0
	QA	**0.42**	**0.42**	0.08	0	0.08	0	0	0
	API	**0.50**	**0.50**	0	0	0	0	0	0
(b) Task 2									
Source states	Eclipse	0	0.12	0.26	**0.28**	0.15	0.02	0.01	0.15
	SE	0.19	0	0.26	0.06	0.14	0.03	0.05	**0.28**
	TT	**0.58**	0.25	0	0.03	0.05	0	0.02	0.07
	DS	**0.81**	0.09	0.04	0	0.02	0.02	0	0.02
	TF	**0.56**	0.25	0.08	0.02	0	0.01	0.01	0.08
	CH	**0.45**	0.31	0.03	0.07	0.03	0	0.07	0.03
	QA	0.20	**0.33**	0.10	0	0.10	0.13	0	0.13
	API	**0.53**	0.27	00.10	0.02	0.02	0.01	0.04	0

5.7 Implications of Behavior Analysis Results

The time-series HCI data extracted from screen-captured task videos by the *scvRipper* tool enables the study of the developers' micro-level behavior 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 in software development.

5.7.1 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. Inferring 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 raises 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.

5.7.2 *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.

5.7.3 *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 become less effective, 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 share 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.

5.8 Threats to Validity

A major threat to the internal validity of our case study is that many findings in this study are based on the time-series HCI data extracted by the *scvRipper* tool. We evaluate the robustness of the *scvRipper* tool using randomly sampled 500 screenshots. Furthermore, to develop confidence in the quality of the extracted HCI data and the analysis results, we manually examine all the analysis results against the task videos. However, such manual examination is qualitative in nature. It is almost impossible to quantitatively examine all the extracted HCI data against the task videos.

A major threat to the external validity of the study is that we study developers' behavior in a controlled experiment instead of a real-world context. Although the study involves two realistic tasks, the complexity of the software to be developed or maintained may limit the generalizability of our findings. Furthermore, due to the limited number of developers and their limited diversity, our study is exploratory by nature. Further studies are required to generalize our findings.

6 Discussion

In this case study, we demonstrate that our *scvRipper* tool supports the quantitative analysis of developers' micro-level behavioral patterns when they interleave coding and web search in software development. *scvRipper* could help the similar human studies in software engineering. In Section 2, we analyze video coding levels in priori studies on human aspects of software engineering. Developer actions and artifacts have to be transcribed in order to perform quantitative analysis of developer behavior in different software development tasks. Our *scvRipper* tool can significantly ease the process of video analysis (e.g., open coding) in such quantitative analysis by automatically extracting application usage and application content from the screen-captured video. Furthermore, as our case study demonstrates, the extracted data can be programmatically abstracted and aggregated, which could provide quantitative evidence to the qualitative observation of the video data.

Based on the extracted HCI data, some actions can be automatically inferred from the context of the recognized application windows, for example, switching between applications, switching between documents within an application, reading code in Eclipse editor, navigating through Eclipse views. However, due to the limitation of computer vision techniques, *scvRipper* cannot reliably detect fine-grained developer actions, such as code editing, setting breakpoints, text selection, and window scrolling. Indeed, through the development of the *scvRipper* tool, we find that it is very difficult to use computer-vision techniques to track such fine-grained mouse and keyboard actions. This finding leads to the development of our *ActivitySpace* framework (Bao et al. 2015a, d) that combines computer vision technique with operating-system level mouse and keyboard instrumentation. Through this combination, our *ActivitySpace* framework can track and analyze fine-grained developer actions in their daily work.

7 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 is a live application, for example, hove 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 a 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 require the support of sophisticated reflection APIs (e.g., Accessibility API or UI Automation API) provided by applications, operating systems and GUI toolkits. Furthermore, a person can use several software tools (e.g., Eclipse IDE, different web browsers) in his work. Instrumenting all these software tools require significant efforts.

Some work proposes to combine low-level operating system APIs and computer vision techniques to track human computer interaction. Hurst et al. (2010) leverages image differencing and template matching techniques to improve the accuracy of target identification that the users click. 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.

Although operating-system instrumentation can easily track fine-grained user action, one difficulty in using operating-system instrumentation data is that the data is too fine-grained to infer higher-level user actions. Our recent work (Bao et al. 2015a, d) present a remembrance framework *ActivitySpace* that combines the computer vision techniques that *scvRipper* implements with operating-system instrumentation. The screenshots provide application context to understand and analyze operating-system instrumentation data. Another interesting extension could be integrating speech recognition techniques to analyze audio data collected by think-aloud method (Mavrikis et al. 2014). This audio data could provide more contextual information regarding the developers' intentions and strategies.

8 Conclusions and Future Work

This paper presented a computer-vision based video-scraping technique (called *scvRipper*) that can automatically extract time-series HCI data from screen-captured videos. We revealed the need for such an automatic video scraping technique through a formative study of the challenges in manual transcription and coding of screen-captured videos. Our *scvRipper* 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 robustness of the *scvRipper* tool, and identified the bottleneck of the tool's runtime performance and suggested potential solutions. The fine-grained quantitative analysis of the task videos of developers' online search behavior in software development demonstrated the usefulness of the extracted time-series HCI data in modeling and analyzing the developers' micro-level behavioral patterns during software development.

Recently, we managed to collect over 2,000-hours working data of 58 developers from a software development company using our recent data collection framework *ActivitySpace* (Bao et al. 2015a, d), a tool built on the core *scvRipper* techniques and operating-system level instrumentation. We are in the process of analyzing this large scale data to mine strategies of professional developers for online search, program comprehension, testing, and software maintenance.

Acknowledgments This work was partially supported by the Major State Basic Research Development Program of China (973 ProgramNo.2015CB352201) and National Key Technology R&D Program of the Ministry of Science and Technology of China (No. 2013BAH01B01). This work is supported by NTU SUG M4081029.020 and MOE AcRF Tier1 M4011165.020.

References

Ammar N, Abi-Antoun M (2012) Empirical evaluation of diagrams of the run-time structure for coding tasks. In: Proceedings of the WCRE, pp 367–376

Bandura A (1986) Social foundations of thought and action: a social cognitive theory, vol 1, p 617

Banovic N, Grossman T, Matejka J, Fitzmaurice G (2012) Waken: reverse engineering usage information and interface structure from software videos. In: Proceedings of the UIST, pp 83–92

Bao L, Ye D, Xing Z, Xia X (2015a) ActivitySpace: a remembrance framework to support interapplication information needs. In: Proceedings 30th IEEE/ACM international conference on automated software engineering

Bao L, Li J, Xing Z, Wang X, Zhou B (2015b) Reverse engineering time-series interaction data from screen-captured videos. In: Proceedings of the SANER, pp 399–408

Bao L, Li J, Xing Z, Wang X, Zhou B (2015c) scvRipper: video scraping tool for modeling developers behavior using interaction data. In: Proceedings of the ICSE, pp 673–676

Bao L, Xing Z, Wang X, Zhou B (2015d) Tracking and analyzing cross-cutting activities in developers' daily work. In: Proceedings of the 30th IEEE/ACM international conference on automated software engineering

Barabasi AL (2005) The origin of bursts and heavy tails in human dynamics. Nature 435(7039):207–211

Bateman S, Teevan J, White RW (2012) The search dashboard: how reflection and comparison impact search behavior. In: Proceedings of the CHI, 1785

Bay H, Ess A, Tuytelaars T, Van Gool L (2008) Speeded-up robust features (surf). Comp Vision Image Underst 110(3):346–359

Bordes A, Gabrilovich E (2014) Constructing and mining web-scale knowledge graphs: KDD 2014 tutorial. In: Proceedings of the KDD, p 1967

Brade K, Guzdial M, Steckel M, Soloway E (1992) Whorf: A visualization tool for software maintenance. In: Proceedings 1992 IEEE workshop on visual languages, pp 148–154

Brandt J, Guo PJ, Lewenstein J, Dontcheva M, Klemmer SR, Francisco S (2009) Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In: Proceedings of the CHI, pp 1589–1598

Brandt J, Dontcheva M, Weskamp M, Klemmer SR, Francisco S (2010) Example-centric programming: integrating web search into the development environment. In: Proceedings of the CHI, pp 513–522

Brown JS, Collins A, Duguid P (1989) Situated cognition and the culture of learning

Canny J (1986) A computational approach to edge detection. IEEE Trans Pattern Anal Mach Intell:679–698

Chang T-H, Yeh T, Miller R (2011) Associating the visual representation of user interfaces with their internal structures and metadata. In: Proceedings of the UIST, pp 245–256

Chang T-H, Yeh T, Miller RC (2010) GUI testing using computer vision. In: Proceeding of the CHI, pp 1535–1544

Colin Cameron A, Windmeijer FA (1997) An r-squared measure of goodness of fit for some common nonlinear regression models. J Econ 77(2):329–342

Corritore CL, Wiedenbeck S (2000) Direction and scope of comprehension-related activities by procedural and object-oriented programmers: An empirical study. In: Proceedings of the IWPC. IEEE, pp 139–148

Corritore CL, Wiedenbeck S (2001) An exploratory study of program comprehension strategies of procedural and object-oriented programmers. Int J Hum-Comput St 54(1):1–23

Dekel U, Herbsleb JD (2009) Reading the documentation of invoked API functions in program comprehension, pp 168–177

Dewan P, Agarwal P, Shroff G, Hegde R (2009) Distributed side-by-side programming. In: Proceedings of the 2009 ICSE workshop on cooperative and human aspects on software engineering, pp 48–55

Dixon M, Fogarty J (2010) Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In: Proceedings of the CHI, pp 1525–1534

Duala-Ekoko E, Robillard MP (2012) Asking and answering questions about unfamiliar APIs: an exploratory study. In: Proceedings of the ICSE, pp 266–276

Ester M, Kriegel H-P, Sander J, Xu X (1996) A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of the KDD, vol 96, pp 226–231

Forsyth DA, Ponce J (2002) Computer vision: a modern approach. Prentice Hall Professional Technical Reference

Fritz T, Shepherd DC, Kevic K, Snipes W, Bräunlich C (2014) Developers' code context models for change tasks. In: Proceedings of the FSE, pp 7–18

Gonzalez RC, Woods RE (2002) Digital image processing. Prentice hall Upper Saddle River, NJ

Greiler M, van Deursen A, Storey M (2012) Test confessions: a study of testing practices for plug-in systems, pp 244–254

Guha R, Guha R, McCool R, McCool R, Miller E, Miller E (2003) Semantic search. In: Proceedings of the WWW, pp 700–709

Hartmann B, Macdougall D, Brandt J, Klemmer SR (2010) What would other programmers do? Suggesting solutions to error messages. In: Proceedings of the CHI, pp 1019–1028

Hilbert DM, Redmiles DF (2000) Extracting usability information from user interface events. ACM Comput Surv 32(4):384–421

Hundhausen CD, Brown JL, Farley S, Skarpas D (2006) A methodology for analyzing the temporal evolution of novice programs based on semantic components. In: Proceedings of the ACM international computing education research workshop, pp 59–71

Hurst A, Hudson SE, Mankoff J (2010) Automatically identifying targets users interact with during real world tasks. In: Proceedings of the IUI. ACM, pp 11–20

Kim JH, Gunn DV, Schuh E, Phillips B, Pagulayan RJ, Wixon D (2008) Tracking real-time user experience (TRUE): a comprehensive instrumentation solution for complex systems. In: Proceedings of the CHI, pp 443–452

Kimble C, Hildreth PM, Bourdon I (2008) Communities of practice: creating learning environments for educators, vol 1. Information Age Publisher

Ko AJ, Myers BA (2004) Designing the whyline: a debugging interface for asking questions about program behavior. In: Proceedings of the CHI, pp 151–158

Ko AJ, Myers BA (2005) A framework and methodology for studying the causes of software errors in programming systems. J Visual Lang Comput 16(1):41–84

Ko AJ, Aung HH, Myers BA (2005a) 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, pp 1557–1560

Ko AJ, Aung HH, Myers BA (2005b) Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In: Proceedings of the ICSE, pp 126–135

Ko AJ, Myers BA, Coblenz MJ, Aung HH (2006) An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans Softw Eng 32(12):971–987

Koru AG, Ozok A, Norcio AF (2005) The effect of human memory organization on code reviews under different single and pair code reviewing scenarios. ACM SIGSOFT Software Engineering Notes 30:1–3

Kushman N, Katabi D (2010) Enabling configuration-independent automation by non-expert users. In: Proceedings of the ninth USENIX symposium on operating systems design and implementation, pp 223–236

Lawrance J, Bellamy R, Burnett M, Rector K (2008) Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In: Proceedings of the CHI. ACM, pp 1323–1332

Lawrance J, Bogart C, Burnett M, Bellamy R, Rector K, Fleming SD (2013) How programmers debug, revisited: an information foraging theory perspective. IEEE Trans Softw Eng 39(2):197–215

Lawrence J, Clarke S, Burnett M, Rothermel G (2005) How well do professional developers test with code coverage visualizations? An empirical study. In: Proceedings of the VL/HCC, pp 53–60

Leary MR (1991) Introduction to behavioral research methods. Wadsworth Publishing Company

Li H, Xing Z, Peng X, Zhao W (2013) What help do developers seek, when and how? In: Proceedings of the WCRE, pp 142–151

Lowe DG (1999) Object recognition from local scale-invariant features. In: Proceedings of the ICCV, vol 2, pp 1150–1157

Matejka J, Li W (2009) CommunityCommands: command recommendations for software applications. In: Proceedings of the UIST, pp 193–202

Matejka J, Grossman T, Fitzmaurice G (2011) Ambient help. In: Proceeding of the CHI, pp 2751–2760

Mavrikis M, Grawemeyer B, Hansen A, Gutierrez-Santos S (2014) Exploring the potential of speech recognition to support problem solving and reflection. In: Open learning and teaching in educational communities. Springer, Berlin, pp 263–276

Muja M, Lowe DG (2009) Fast approximate nearest neighbors with automatic algorithm configuration. In: VISAPP (1), pp 331–340

Murphy-Hill ER, Zimmermann T, Nagappan N (2014) Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development? In: Proceeding of the ICSE, pp 1–11

Nie Z, Zhang Y (2005) Object-level ranking: bringing order to web objects. In: Proceeding of the WWW, pp 567–574

Nie Z, Ma Y, Shi S, Wen J-r, Ma W-y (2007) Web Object Retrieval. In: Proceeding of the WWW, pp 81–90

Parzen E (1962) On estimation of a probability density function and mode. Ann Math Stat 33(3):1065–1076

Piorkowski D, Fleming SD, Scaffidi C, John L, Bogart C, John BE, Burnett M, Bellamy R (2011) Modeling programmer navigation: a head-to-head empirical evaluation of predictive models. In: Proceeding of the VL/HCC, pp 109–116

Ponzanelli L, Bacchelli A, Lanza M (2013) Seahawk: Stack overflow in the IDE. In: Proceeding of the ICSE, pp 1295–1298

Rabiner L, Juang BH (1986) An introduction to hidden Markov models. IEEE ASSP Mag 3(1):4–16

Rhodes B (1996) 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, pp 122–125

Robillard MP, Coelho W, Murphy GC (2004) How effective developers investigate source code: an exploratory study. IEEE Trans Softw Eng 30(12):889–903

Rosten E, Drummond T (2006) Machine learning for high-speed corner detection. In: Computer Vision–ECCV 2006. Springer, Berlin, pp 430–443

Sarma A, Maccherone L, Wagstrom P, Herbsleb J (2009) Tesseract: interactive visual exploration of socio-technical relationships in software development. In: Proceeding of the ICSE, pp 23–33

Sawadsky N, Murphy GC (2011) Fishtail: from task context to source code examples. In: Proceeding of the 1st workshop on Developing tools as plug-ins - TOPI, p 48

Shi J, Malik J (2000) Normalized cuts and image segmentation. IEEE Trans Pattern Anal Mach Intell 22(8):888–905

Sillito J, De Voider K, Fisher B, Murphy G (2005) Managing software change tasks: an exploratory study. In: International Symposium on Empirical Software Engineering, IEEE, p 10

Silverman BW (1986) Density estimation for statistics and data analysis, vol 26. CRC press

Sinha SN, Frahm J-M, Pollefeys M, Genc Y (2006) GPU-based video feature tracking and matching. In: EDGE, workshop on edge computing using new commodity architectures, vol 278, p 4321

Vakilian M, Chen N, Negara S, Rajkumar BA, Bailey BP, Johnson RE (2012) Use, disuse, and misuse of automated refactorings. In: Proceeding of the ICSE, pp 233–243

von Mayrhauser A, Vans AM (1997) Program understanding behavior during debugging of large scale software. In: Empirical Studies of Programmers, 7th Workshop, ACM. ACM, pp 157–179

Wang J, Peng X, Xing Z, Zhao W (2011) An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. In: Proceeding of the ICSM, pp 213–222

Wang J, Peng X, Xing Z, Zhao W (2013) Improving feature location practice with multi-faceted interactive exploration. In: Proceeding of the ICSE, pp 762–771

Weisstein EW (2011) Least squares fitting–exponential. MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/LeastSquaresFittingExponential.html

Whittaker JA, Poore JH (1993) Markov analysis of software specifications

Wu D-C, Tsai W-H (2000) Spatial-domain image hiding using image differencing. Proc ICCVISP 147(1):29–37

Yang J, McAuley J, Leskovec J, LePendu P, Shah N (2014) Finding progression stages in time-evolving event sequences. In: Proceeding of the WWW, pp 783–794

Yeh T, Chang T-H, Miller RC (2009) Sikuli: using GUI screenshots for search and automation. In: Proceeding of the UIST, pp 183–192

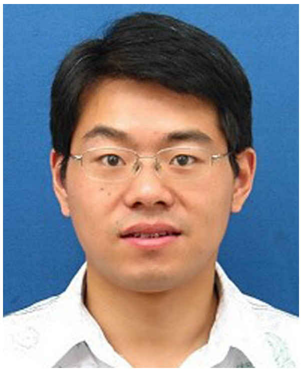
Zhang Q, Chen Y, Zhang Y, Xu Y (2008) SIFT implementation and optimization for multi-core systems. In: Proceeding of the IPDPS, pp 1–8

Zhu J, Nie Z, Liu X, Zhang B, Wen J-R (2009) StatSnowball: a statistical approach to extracting entity relationships. In: Proceeding of the WWW, p 101

Lingfeng Bao is currently a PhD candidate in the College of Computer Science and Technology, Zhejiang University. He received his B.E. from the College of Software Engineering, Zhejiang University in 2010. His research interests include software analytics, behavioral research methods, data mining techniques, and human computer interaction.

Jing Li is currently a PhD candidate in Computer Science, at the School of Computer Engineering, Nanyang Technological University (NTU), Singapore. He received his B.E. and M.E. in Electrical Engineering from University of Electronic Science and Technology of China (UESTC), China, in 2010 and 2013, respectively. His research aims to understand, predict, and enhance developer behavior in software development and maintenance, in social and information networks by developing techniques in machine learning, data mining, network analysis and natural language processing.

Zhenchang Xing is the Assistant Professor at the School of Computer Engineering, Nanyang Technological University, Singapore. Dr. Xing's research interests include software engineering and human-computer interaction. His work combines software analytics, behavioral research methods, data mining techniques, and interaction design to understand how developers work, and then build recommendation or exploratory search systems for the timely or serendipitous discovery of the needed information.



Xinyu Wang received the BE and PhD degrees from Zhejiang University, Hangzhou, China, in 2002 and in 2007. He is an Associate Professor in the College of Computer Science and Technology, Zhejiang University. His primary research interests include software engineering, distributed software architecture, and distributed computing.

Xin Xia received his PhD degree in computer science from the College of Computer Science and Technology, Zhejiang University, China in 2014. He is currently a research assistant professor in the college of computer science and technology at Zhejiang University. His research interests include software analytic, empirical study, and mining software repository. He is a member of the Institute of Electrical and Electronics Engineers.

Bo Zhou received the PhD degree from Zhejiang university in 1996. He is a professor in the college of computer science and technology at Zhejiang university. His research interests include database management system, distributed computing and software engineering.