UI-Tracer: A Lightweight Approach to Help Developers Tracing User Interface Elements to Source Code

Regina Hebig[1]

Abstract: The ability to understand software systems is crucial to identify hidden threats or maintain software systems over many years. Still software comprehension activities take up around 58% of software development time. While most approaches support the comprehension of a software system's code perspective, its connection to the user perspective is barely explored. We present UI-Tracer, a lightweight support for tracing user interface elements to source code using the version history of a system. The evaluation on two open source systems shows that the approach can cover all UI elements that have been changed or added within the accessible part of the version history. Furthermore, the median numbers of files flagged as potentially responsible for a UI element is 8 and 3 for the two studied systems. Thus, UI-Tracer provides an easy starting ground for developers to identify files relevant for future UI changes.

Keywords: Software Comprehension; UI Tracing

1 Introduction

One of the key ideas behind open source systems is that they enable users and experts to inspect the source code, making it more difficult to build in hidden threats. Despite the openness it is, however, not trivial to read, comprehend or explain source code. Even experienced software developers have to invest time if they want to understand a new system. Also systems with long live-spans need to be understood over the years over and over again by new developers and maintainers. While there are approaches that can be used to easier understand a software system, until today, software comprehension activities take up around 58% of software developers' time [Xi17, Fj83, Ti11]. Also, developers who know a system have little tool support, when explaining the source code to new developers.

Related Work Most existing works towards software comprehension can be split into code-level and architecture level approaches. Thus, there are coding conventions, e.g. Oracle's Java Coding Conventions[2], code annotations, as well as approaches for stepwise execution/simulation, e.g. [DL00] and debugging, e.g. [ZL96]. Some approaches aim at automatically recovering trace links in source code [An02]. Furthermore, there is the

[1] Chalmers | University of Gothenburg Universität, Software Engineering Division, Hörsegången 11, 412 96 Göteborg, Sweden hebig@chalmers.se

[2] Oracle's Java Coding Conventionshttp://www.oracle.com/technetwork/java/index-135089.html

research area of code visualizations, that focuses on illustrating metrics about building blocks, such as number of lines of code or complexity, as well as relations between these building blocks. Examples are code cities [WL08] and circular bundle views [TBD12]. Other approaches aim at reverse engineering models from code [Ch08], generation of documentations, e.g. with Doxygen[3] and approaches to summarize software in natural language [Bi13]. Nearly all of these techniques remain at representing the source code perspective on the system. The resulting visualizations, simulations, models, documentation, and explanations, reflect the structure of the code and its terminology, such as class names. However, it can be argued that each software system has another fundamental perspective: the user perspective. So far there is no approach that provides a bridge between user and source code perspective.

UI-Tracer Research has shown that professional developers who have to comprehend a new piece of code try to connect knowledge about the user interface with knowledge about the code [Ro12]. However, so far there is no approach that supports developers in making these connections. On the other hand, it might take even experienced developers some effort to reproduce and document these traces, when explaining source code to novices.

In this paper, we ask the following questions: How can an automated approach help users to identify source code that has impact on a user interface element of interest? We present the UI-Tracer, a lightweight approach that analyzes the UI of a system throughout its version history, and identifies files that were changed together with observed changes in the UI. This way the UI-Tracer can provide developers who want to learn about the code connected to an UI element, with a starting set of files. While not necessarily all of the files are relevant for the UI element, the value lies in the limitation of choices, making it easier to find the right files. We evaluate the approach based on two open source systems with regards to the questions a) whether all UI elements can be covered with the approach and b) how small, i.e. precise, are the sets of identified files. The UI-Tracer approach is presented in Section 2. In Section 3, we evaluate the approach and discuss the results. Finally, in Section 4 we conclude and discuss future work.

2 The UI-Tracer

The basic idea of the UI-Tracer[4] approach is that the following two questions have the same answers: "What code is responsible for a UI element?" and "What code causes changes of a UI element when changed?". Thus, when tracing UI elements to code, those classes are of interest that can change a UI element. Furthermore, change happens during the development of a system, where each element is at one point or another introduced.

Therefore, this approach uses the granularity of commit to a project for the tracing. Analyzing how UI and source code change commit by commit. The three technical pillars of this

[3] Doxygen `http://www.stack.nl/~dimitri/doxygen/`

[4] The UI-Tracer prototype can be found here `https://github.com/rhebig/UI-Tracer`

approach are: online open source repositories, e.g. GitHub, technologies for automatic compilation, e.g. ANT, and user interface scripting languages and image comparison techniques, such as Sikuli [YCM09]. In the following subsections it will be explained how the UI-Tracer works and how it can be adapted to analyzing new systems.

2.1 Approach and Prototype

Figure 1 provides an overview of the approach, which consists of two halves. A class diagram of the UI-Tracer can be seen in Figure 2.

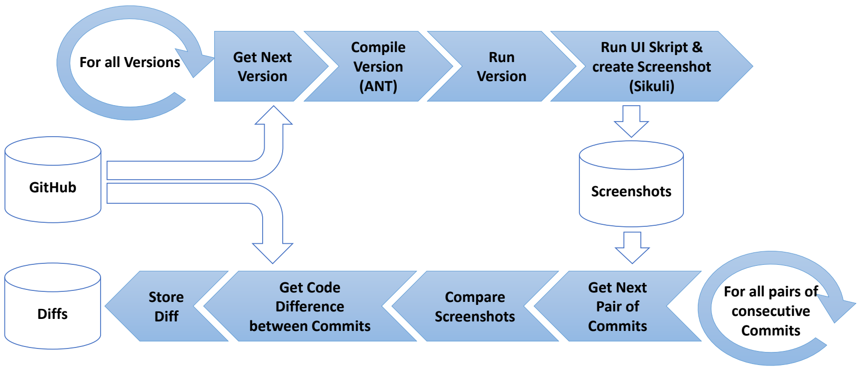


Fig. 1: Overview about the approach

In the ***first half***, the repository of the system to be analyzed is cloned and then commit by commit reverted back to former versions (performed by the class *CloneRemoteRepository* shown in Figure 2). For each commit/version, the UI-Tracer compiles the software to create an executable, e.g. a jar (performed by the class *Builder* shown in Figure 2). This is done at the moment for Java using Ant. However, since, the build process is the only step the approach that is specific to the programming language. In future, the prototype can be extended at this point to allow for other technologies, such as Gradle or Maven, and with it further programming languages, e.g. C++ or python.

After the executable is created, the UI-Tracer (class *BuildAndScreenshotCoordinator* shown in Figure 2) uses a process builder to execute the compiled system. Once the system is running the UI-Tracer executes some random mouse movement. This is done to make sure that random UI effects, as they sometimes occur with just started systems, disappear. Then a small standard routine is executed to enlarge the systems window to full screen. This is done to ensure that the collected images are comparable. Afterward the first screen-shot is done.

In the next step a customized Sikuli script is executed. This script is typically created based on the most current version of the system and can for example click and hover over all main menu items to make sure that the sub-menu items are shown as well. During that execution extra screen-shots are taken and stored. It turned out to be a crucial step to terminate the running system after making all relevant screen-shots. Otherwise the machine will be overloaded quickly and slow down dramatically. Therefore, the UI-Tracer kills the started process before downloading a new version of the system from Github.

In the ***second half*** the class *ComparingScreens* (Figure 2) iterates over all pairs of consecutive commits. Note that it is possible that single versions cannot be compiled. One possible reason for that is that a commit in the repository introduced faulty or breaking code, which was corrected in a later commit. To compensate for that two commits are considered consecutive, if screen-shots could successfully be obtained for both commits and if there are no commits in between them for which screen-shots could be applied.

For each pair, the initial screen-shots are subject to an automated image comparison. Similarly, for each screen-shot made during the execution of the customized Sikuli script a comparison between the two commits is performed. If a pair of screen-shots is marked as different, by the automated comparison, the UI-Tracer will start to use GitHub to retrieve the code difference between the involved commits (performed by class *CodeDiffTracer* shown in Figure 2). In doing so, the UI-Tracer considers all changes of all commits that took place in between the considered pair of commits (to cover for cases where some commits could not be successfully compiled). Finally, the ids of those commits are stored together with references to the two screen-shots and the list of files in the repository that have been modified, added, or deleted in between the two commits.

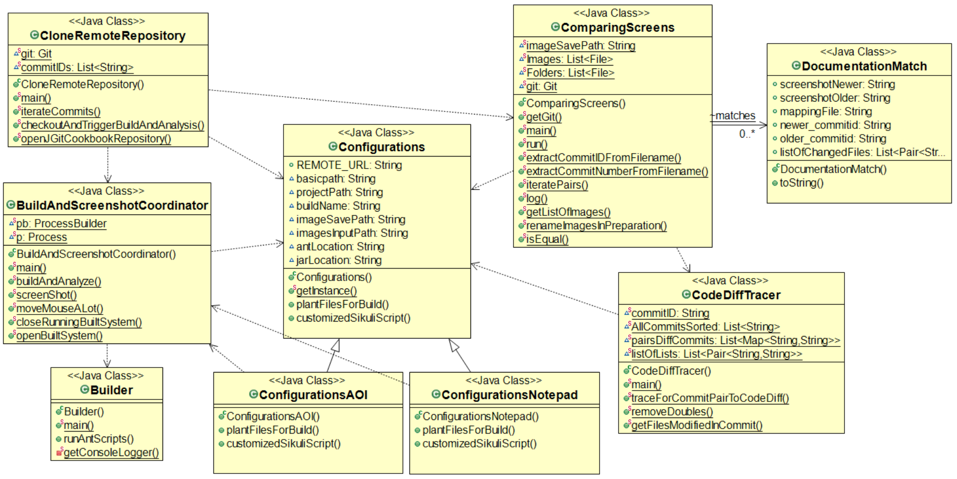


Fig. 2: Class Diagram of the UI-Tracer

2.2 Configuring the Tool to Analyze a new Software System

At the moment the UI-Tracer uses a primitive configuration class to enable its application to new systems. For a new system to be studied a new class can be created that inherits from the class *Configurations* (as shown in Figure 2 for the two example systems AOI and Notepad). Then the *getInstance()* method in class *Configurations* should be changed to return the instance of the newly created class.

Plant Files for Build While some systems already provide an automated Ant script, others do not. Therefore, it might be necessary to plant the Ant script and used libraries, e.g. jar files, into the system. Depending on the system, very old version might have slight differences that need to be addressed in the Ant script, e.g. a differently named main class. A typical implementation of the plantFilesForBuild(Integer buildCount) method is shown in Listing 1. Here another Ant script is planted when the commit to be built is 80 commits older than the current version (or more). The Ant script and relevant libraries are thereby stored in a folder *FilesToPlant* of which all content is copied to the system to be compiled.

```
public void plantFilesForBuild(Integer buildCount) {
        File srcDir;
        if(buildCount<80)
                srcDir = new File(basicpath + "FilesToPlant");
        else
                srcDir = new File(basicpath + "FilesToPlant_2");
        File destDir = new File(projectPath);
        try {
                FileUtils.copyDirectory(srcDir, destDir);
        }catch(Exception e) {System.out.println(e);}
}
```

List. 1: Typical implementation of plantFilesForBuild

Customized Sikuli Script The customized sikuli script can be added here. Listing 2 shows an example of such a script with a typical building block. The parameters *screenRect*, *commitNumber*, and *commitID* are there to ensure that the screen-shot captures the relevant region of the screen and that it is saved in association to the current commit. These parameters are only used when the *BuildAndScreenshotCoordinator* is called to take a screen-shot and can otherwise be ignored. When calling the latter a fourth parameter should be used to define a subcategory of screen-shots, so that the automated comparison can later on compare the right screen-shot with each other. In the example, the subcategory is set to *DropDown* by defining the folder “DropDown\\” as target for saving the screen-shot.

```
public void customizedSikuliScript(Rectangle screenRect, Integer
    commitNumber, String commitID) {
        Screen s = new Screen();
        try {
                s.click(imagesInputPath+"DropDown.png");
                TimeUnit.SECONDS.sleep(1);
                BuildAndScreenshotCoordinator.screenShot(screenRect,
                    commitNumber, commitID, "DropDown\\");
                s.type(Key.ESC);
        }catch(Exception e) {System.out.println(e);}
}
```

List. 2: Typical building block of the customized sikuli script

Tab. 1: Paths to be set

Variable	
REMOTE_URL	the url of the repository that contains the system that should be analyzed
basicpath	a basis path to the working directory
projectPath	(extension to basicpath) the path where the system will be cloned to
buildName	the name of the build
imageSavePath	(extension to basicpath) the path to the folder where screen-shots will be stored
imagesInputPath	(extension to basicpath) the path to the folder where images are stored that are used in the customized sikuli script
antLocation	(extension to projectpath) the path to the xml file specifying the ant script
jarLocation	(extension to projectpath) the path to the executable that will be created when running Ant

Another important aspect is that Sikuli allows users to define the target of a click or hover action by providing an image of the region or button. By matching the image to the screen, Sikuli is robust against factors, such as absolute positioning of user interface elements. Figure 3 shows the image used in the example. Other actions that are useful are sleeping actions to give the program time to react to an action, e.g. after a click, and escape actions to go back to the starting screen, e.g. by pressing the ESC key programmatically. This way the script can also be used to navigate between multiple views of a UI.

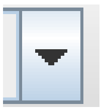


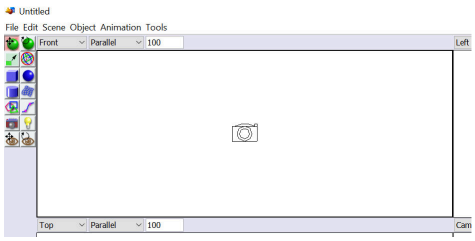
Fig. 3: Image of the DropDown to be clicked in listing 2

Paths and URLs Finally, the constructor of the new subclass of *Configurations* should set all relevant paths and URLs, as listed in Table 1.

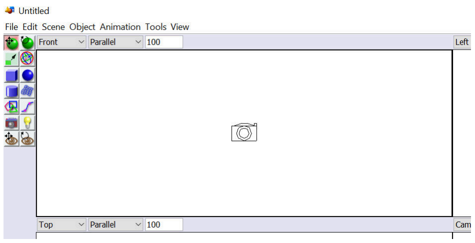
2.3 Example Output and Envisioned Use

An example for the output can be seen in Figures 4 and 5. The figures show parts of screen-shots that UI-Tracer made for the Art Of Illusion (AOI) software, compiled 77 commits before the current version and 78 commit before the current version. Note, that screen-shots from the initial screen show that a new menu item was added in commit 77 that has not yet been there after commit 78: *View*. Furthermore, the Figure shows the screen-shots done after clicking/hovering over the buttons *Scene* and *View*. Observe that several sub-menu items from the menu item *Scene* in commit 78 are moved to the new menu item *View*. UI-Tracer reports that only one file was modified in commit 77: “src/artofillusion/LayoutWindow.java”. Thus, a user of UI-Tracer can conclude that the definition of menu items and sub-menu items is done in that file.

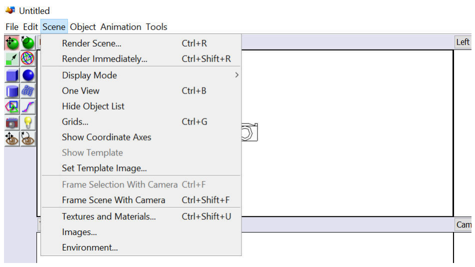
The current version of the UI-Tracer can be used by developers who are interested in learning about an open source system’s implementation, e.g. in order to make future contributions. The images can be used as an entry point to identify what files are responsible for certain UI elements.



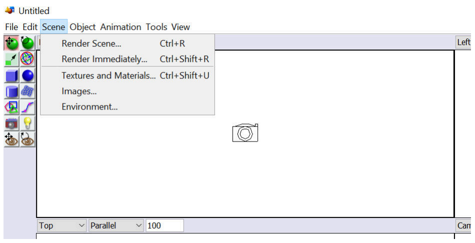
(a) AOI 78 commits before current version, initial screen



(b) AOI 77 commits before current version, initial screen


Untitled
File Edit Scene Object Animation Tools
Render Scene... Ctrl+R
Render Immediately... Ctrl+Shift+R
Display Mode
One View Ctrl+B
Hide Object List
Grids... Ctrl+G
Show Coordinate Axes
Show Template
Set Template Image...
Frame Selection With Camera Ctrl+F
Frame Scene With Camera Ctrl+Shift+F
Textures and Materials... Ctrl+Shift+U
Images...
Environment...
Left
Cam

(c) AOI 78 commits before current version, clicking *Scene* menu


Untitled
File Edit Scene Object Animation Tools View
Render Scene... Ctrl+R
Render Immediately... Ctrl+Shift+R
Textures and Materials... Ctrl+Shift+U
Images...
Environment...
Left
Top Parallel 100 Cam

(d) AOI 77 commits before current version, clicking *Scene* menu

Fig. 4: Example of automatically detected difference, between screen-shots of AOI consecutive commits 77 and 78 (i.e. 77 and 78 commits before the current version).

Another possible use case is to apply the UI-Tracer incrementally during development that follows the continuous integration principle. This could be done by hooking it into the repository and automatically executing it with every new commit. In both cases the resulting images (e.g as in Figure 4), can be integrated into the documentation of the system, to give readers a quick and intuitive impression about what a class, such as *LayoutWindow*, is responsible for.

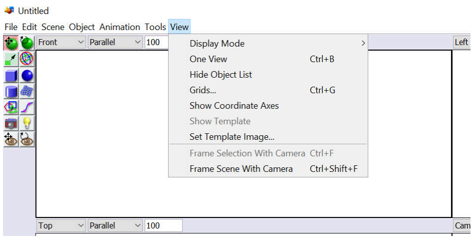


Fig. 5: Continuation of example of automatically detected difference: AOI 77 commits before current version, clicking *View* menu

3 Evaluation

We evaluate the two following questions:
(a) Does the approach cover all UI elements?
(b) How small, and with it useful, are the sets of identified files?
The following subsections describe the used case systems for the evaluations, the method used for data collection, the results of the evaluation as well as a discussion of these results.

3.1 Used Systems

To evaluate the approach it was applied to two open source system: PHNotepad[5], which is a simple text editor written in Java, and Art of Illusion[6], a program to create 3D models.

PHNotepad is a java project with 8 contributors. The fork[7] that was analyzed was made on the 11th of October 2017 and had 85 commits on GitHub. The oldest commit has been submitted in May 2012 and included an initial small version of the system with 15 files. AOI is a java project as well and has 4 contributors. UI-Tracer was used to analyze a fork of AOI[8] made on the 12th of September 2017, when the project had 526 commits on github. The project was developed already before it came to GitHub. Therefore, AOI had already reached version 2.4.1 when it was initially committed to Github in March 2007. The last stable release from December 2016 is version 3.0.3.

3.2 Method

UI-Tracer was configured for each of the two systems and run over night. For AOI, no ant and jar files needed to be planted for the most recent 537 commits as the developers use Ant themselves. However, for older commits 15 jars and an ant file needed to be provided. Here it was possible to reuse the ant file that is used for the most recent versions. Furthermore, the Ant file needed to be changed slightly for commits older than the last 417 commits, as at that commit the main class and method were renamed. The customized sikuli script was written, so that all main menu items are clicked and the sub-menu items become visible in the respective screen-shots

Also PHNotepad comes with an ant script. However, to ensure that the system compiles properly, a new ant script was planted for each commit. Altogether, three different ant scripts are planted, as in the early version of the system no image files were used and the main class was renamed at some point. Again, the customized sikuli script was written, so that all main menu items are clicked to show the sub-menu items. Furthermore, the sikuli script clicks through all the tool icons at the top row, to make eventual pup-up windows visible.

To make sure that the tool provided correct results, two manual checks were performed. The first manual check made sure that the comparison of the screen-shots worked correctly, i.e. that no difference was missed and no identical screen-shots were identified as different. The second manual check was performed to ensure that the tracing of code diffs for a pair of commits worked correctly.

To get a baseline of UI elements for which changes should be identify by the UI-Tracer, we manually compared the initial commit of each system with its most current version. In this

[5] PHNotepad `https://github.com/pH-7/Simple-Java-Text-Editor.git`
[6] AOI `https://github.com/ArtOfIllusion/ArtOfIllusion`
[7] PHNotepad fork used in evaluation `https://github.com/rhebig/Simple-Java-Text-Editor`
[8] AOI fork used in the evaluation `https://github.com/rhebig/ArtOfIllusion`

Tab. 2: General statistics about the evaluation runs.

	AOI	**PHNotepad**
Duration of analysis run	3h 15 min.	1h 25 min.
Number of commits in Repository	526	85
Number of commits with successful screen-shots	320	78
Number of commits without successful screen-shots	206	7

comparison we considered UI elements that changed or were added to the initial screens of both tools, as well as changed/added UI elements that are visible when executing the actions encoded in the customizable sikuli scripts, e.g. sub-menus and pop-up windows, when clicking and hovering over menu items and buttons. For each of those UI elements, we then searched the results of the UI-Tracer run to see whether the tool identified at least one UI change, e.g. appearance or change of position, for that UI element. For example, the UI-changes observed in Figure 4 concern 10 UI-elements: the menu item “View”, which was added and the 9 sub-menu items that were moved from “Scene” to “View”.

3.3 Results

The analysis of the 526 AOI commits took around 3 hours and 15 minutes, while the analysis of PHNotepad, which has only 85 commits, was done within around 1 hour and 25 minutes. Table 2 shows some general statistics about the analysis of both systems. For 320 of the AOI commits is was possible to successfully derive screen-shots of the running system (around 60% of the commits). For PHNotepad the amount was with 78 commits much higher (around 91%).

(a) Does the approach cover all UI elements? As baseline, between the initial commit and the current version, 36 UI elements of AOI changed and 32 UI elements of PHNotepad changed. These changes include among other additions of top menu items, e.g. “View” in AOI, additions and renames of sub-menu items, additions and removal of buttons and icons, rearrangements of information fields within the UI, moving of sub-menu items amongst top menu items and changing texts. For AOI the UI-Tracer identified 18 pairs of commits in between which the UI changed, including 51 UI changes that affected 39 different UI elements. For PHNotepad is were 22 pairs of commits, including 77 UI changes that affected 33 different UI elements (see Table 3).

First of all, these identified changes covered all of the 36 (AOI) + 32 (PHNotepad) UI elements for which changes were expected. Note this success-rate does not suffer from the fact that no screen-shots could be obtained for many of the commits. In theory, just comparing the initial and current commit is sufficient to get this coverage.

However, by comparing commits in between, the UI-Tracer identified 3 (AOI) + 1 (PH-Notepad) additional changes. These are changes that are not visible when only comparing the first and last version of a program. The reason for that is that changes can cover each

Tab. 3: Comparison of User Interfaces

	AOI	**PHNotepad**
Number of commit pairs associated to identified UI changes	18	22
Number of UI changes identified by UI-Tracer	51	77
UI elements with identified changes	39	33
UI elements expected to change (initial commit vs. current version)	36	32
Coverage of expected changes by identified changes	**100%**	**100%**
UI elements with changes identified in screen-shots of initial screens	16	14
Number of changes visible in screen-shots of initial screens	20	22
Number of redundant change reports (indication in multiple screen-shots)	121	181

other over time, e.g. such as the addition and later renaming of a UI element. Some changes are even done and re-done after a while, leading to information about additional UI elements.

Finally, 20 (AOI) and 22 (PHNotepad) of the changes, affecting 16 and 14 UI elements, could be identified without the customized sikuli script. Furthermore, the tool often reported multiple times on the same UI change, if the affected elements were visible on the initial screen and the customized screen-shots. We got 121 (AOI) and 181 (PHNotepad) of such redundantly reported changes.

(b) How small, and with it useful, are the sets of identified files? While it is not necessary that all commits are successfully analyzed to get a high coverage of UI elements, it influences the precision of the set of identified files. The reason is that missing information about commits in between two commits which show UI differences creates an uncertainty about which commit introduced the observed changes. Consequently, the files changed in those commits need to be integrated to the set of returned candidates.

As summarized in Table 4 AOI consists of around 713 files (including code, image, and some files) and PHNotepad consists of 32 files. For AOI, where only 60% of the commits were covered, the average number of files identified as relevant for an observed UI change is 21 (around 2.9% of all AOI files). Note that there are three UI changes that belong to the same pair of commits: 204 and 265 commits before the current version. Due to this large gap between the two successful commits, UI-Tracer had to associate 109 files with these three changes. Without these outliers, the average number of identified files for AOI is 15.5. That the average is skewed by outliers is also reflected by the low median of 8 files (1.1% of all AOI files). For PHNotepad, the number of files identified as relevant for an observed UI change is with in average 3.96 files (median 3) quite low (around 12% of all PHNotepad files). Figure 6 summarizes these results.

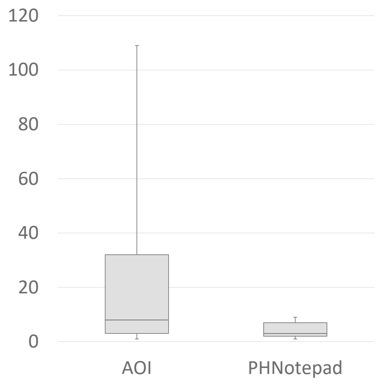


Fig. 6: Files identified by UI-Tracer per found UI changes

Tab. 4: Code Tracing per Observed UI Change

	AOI	**PHNotepad**
absolute number of files in the system	713	32
per Observed UI Change:		
median number of files identified by UI-Tracer	**8**	**3**
average number of files identified by UI-Tracer	21 (15.5 without outliers)	3.96
minimum number of files identified by UI-Tracer	1	1
maximum number of files identified by UI-Tracer	109	9
Number of UI Changes:		
... identified by UI-Tracer	51	77
... with > 2 files identified by UI-Tracer	39	48
... with > 10 files identified by UI-Tracer	23	0

The minimum number of files identified for a UI change is in both cases 1, the maximum numbers are 109 (AOI) and 9 (PHNotepad). Of the 51 identified UI changes in AOI, 12 were associated to 2 or less files and 28 to 10 or less files. Of the 77 PHNotepad UI changes, 29 were associate with 2 or less files and none was associated with more than 10 files.

3.4 Discussion

The results show that it is possible to cover all changes in UI elements that happen within the observed period of time. For those UI elements that are introduced or changed, the approach will associate relevant source code files. Many systems, however, are not open source from the very beginning, such as AOI. Thus, the approach can only cover all elements of the user interface, when applied by someone who has access to the full version history.

Furthermore, the precision of the identified files is dependent on the number of commits that can successfully be compiled and the number of files committed per commit. Both aspects are highly dependent on the project culture.

However, bringing the number of candidate files to look at down to median 3 to 8 files, especially in large systems, such as AOI, is already very useful for users who want to identify responsible parts of the source code. Despite the current limitations the results can be considered as very promising, as they open the path towards a completely new approach of software documentation and comprehension support. The UI-Tracer shows that it is possible to automatically trace UI elements to source code in a lightweight and simple way. This will allow novices to approach new systems by directly connecting the code to the user interface.

4 Conclusion and Future Work

The paper introduced a lightweight approach to support software comprehension, by helping users to trace changes in user interfaces to files in the code base. The evaluation on the two open source systems Art of Illusion and PHNotepad showed that the approach can cover all UI elements that have been changed or added within the accessible part of the version history. With median 3 to 8 files we think that the approach can be a useful support for users who aim at changing or extending a for them unknown system. Furthermore, we think that the UI-Tracer is an important first step towards integrating the user interface perspective with approaches for comprehending source code.

In future work, we aim at further exploring the potentials of the UI-Tracer. The presented approach is in theory independent of the programming language of the system to be analyzed. In future work we plan to adapt UI-Tracer to work with additional build systems, e.g. gradle or Maven, and apply it to systems with different languages. Furthermore, our next step is to explore whether we can achieve a better precision, by automatically planting changes into the source code of a system and observing the changes to the user interface. Another direction for future work is the use of data. For example, AOI is a graphics tool. It will be interesting to see what happens if the customized Sikuli script would not just press buttons but actually use the tool to create graphics.

References

[An02] Antoniol, Giuliano; Canfora, Gerardo; Casazza, Gerardo; De Lucia, Andrea; Merlo, Ettore: Recovering traceability links between code and documentation. IEEE transactions on software engineering, 28(10):970–983, 2002.

[Bi13] Binkley, Dave; Lawrie, Dawn; Hill, Emily; Burge, Janet; Harris, Ian; Hebig, Regina; Keszocze, Oliver; Reed, Karl; Slankas, John: Task-driven software summarization. In: Software Maintenance (ICSM), 2013. IEEE, pp. 432–435, 2013.

[Ch08] Chouambe, Landry; Klatt, Benjamin; Krogmann, Klaus: Reverse engineering software-models of component-based systems. In: Software Maintenance and Reengineering, 2008. CSMR 2008. IEEE, pp. 93–102, 2008.

[DL00] Drappa, Anke; Ludewig, Jochen: Simulation in software engineering training. In: Software Engineering, 2000. IEEE, pp. 199–208, 2000.

[Fj83] Fjeldstad, Richard K: Application program maintenance study: Report to our respondents. Proceedings GUIDE 48, 1983, 1983.

[Ro12] Roehm, Tobias; Tiarks, Rebecca; Koschke, Rainer; Maalej, Walid: How do professional developers comprehend software? In: Proceedings of the 34th International Conference on Software Engineering. IEEE Press, pp. 255–265, 2012.

[TBD12] Trümper, Jonas; Beck, Martin; Döllner, Jürgen: A visual analysis approach to support perfective software maintenance. In: Information Visualisation (IV), 2012. IEEE, pp. 308–315, 2012.

[Ti11] Tiarks, Rebecca: What maintenance programmers really do: An observational study. In: Proceedings of the Workshop Software Reengineering (WSR). pp. 36–37, 2011.

[WL08] Wettel, Richard; Lanza, Michele: Codecity: 3d visualization of large-scale software. In: Companion of the 30th international conference on Software engineering. ACM, pp. 921–922, 2008.

[Xi17] Xia, Xin; Bao, Lingfeng; Lo, David; Xing, Zhenchang; Hassan, Ahmed E; Li, Shanping: Measuring Program Comprehension: A Large-Scale Field Study with Professionals. IEEE Transactions on Software Engineering, 2017.

[YCM09] Yeh, Tom; Chang, Tsung-Hsiang; Miller, Robert C: Sikuli: using GUI screenshots for search and automation. In: Proceedings of the 22nd annual ACM symposium on User interface software and technology. ACM, pp. 183–192, 2009.

[ZL96] Zeller, Andreas; Lütkehaus, Dorothea: DDD - a free graphical front-end for UNIX debuggers. ACM Sigplan Notices, 31(1):22–27, 1996.