Locating User Interface Concepts in Source Code*

Matúš Sulír[1] and Jaroslav Porubän[2]

1 **Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Košice, Slovakia**
`matus.sulir@tuke.sk`

2 **Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Košice, Slovakia**
`jaroslav.poruban@tuke.sk`

Abstract

Developers often start their work by exploring a graphical user interface (GUI) of a program. They spot a textual label of interest in the GUI and try to find it in the source code, as a straightforward way of feature location. We performed a study on four Java applications, asking a simple question: Are strings displayed in the GUI of a running program present in its source code? We came to a conclusion that the majority of strings are present there; they occur mainly in Java and "properties" files.

1998 ACM Subject Classification D.2.3 Coding Tools and Techniques, H.5.2 User Interfaces: Graphical User Interfaces

Keywords and phrases Source code, graphical user interfaces, feature location

Digital Object Identifier 10.4230/OASIcs.SLATE.2016.6

1 Introduction

Developers understand a program only when they are able to mentally connect structures in the program with real-world concepts [2]. Naturally, this connection can be established much more easily if the vocabulary used in the source code resembles the domain terms displayed in the GUI of a program.

One of the most frequent activity performed by a programmer is feature location – finding an initial source code location implementing a given functionality [5]. To perform it, developers rarely use complicated feature location tools and plugins [8], and rely on simple textual search instead [4].

Consider a developer trying to fix a bug in a program he does not know. He will probably start with an exploration of a running UI (user interface) relevant to the bug. He will start to concentrate on particular GUI items, like buttons and menu items causing the bug to manifest. Then, he will try to search for the labels of these GUI items (button captions, menu names) in the source code of the program, using standard search functionality of an IDE (integrated development environment).

The GUI of a program is displayed to an end user – often a paying customer. For this reason, it must contain terms from the problem domain. On the other hand, the source code is rarely shown to a customer. The use of correct domain concepts in the source code is only a recommended practice, often not enforced.

* This work was supported by the FEI TUKE Grant no. FEI-2015-16 "Evaluation and metrics of domain usability".

Table 1 The applications used in the study.

Application	*Java LOC*	*GUI strings*	*GUI words*
ArgoUML 0.34	195,363	307	2,391
FreeMind 1.0.1	67,357	353	1,050
PDFsam 2.0.0	23,774	65	168
Weka 3.6.13	275,036	592	904

We formulate our main hypothesis and two smaller research question for this paper as follows:

- *Hypothesis*: Strings and concepts displayed in the GUI of a running program are located in its static source code, too.
- *RQ1*: When yes, mainly in what types of files are these strings located?
- *RQ2*: If no, what are the most common reasons?

2 Method

To prove our hypothesis, we will automatically extract strings from a running GUI of a few applications and try to search for these strings (and their parts) in the source code of the corresponding program.

In Table 1, there is a summary of the studied objects. We selected three desktop Java applications from the SF110 [6] corpus of open source projects: FreeMind[1] is mind-mapping software, PDFsam[2] splits and merges PDF files, and Weka[3] is machine learning software. Additionally, ArgoUML[4] – a UML modeling tool – was selected as a popular, medium-sized project. The "Java LOC" column in Table 1 denotes the number of source code lines in Java files, measured by the the CLOC[5] program.

2.1 GUI Scraping

Before running the experiment, we ensured English localization was set in all applications, since the source code is written in English and a mismatch between the code and GUI language would produce skewed results. In the case of the FreeMind application, language adjustment in the settings was necessary, all other programs had the language already set correctly.

Every application was fed to a GUI ripper [10] which is a part of the project GUITAR [12]. The GUI ripper fully automatically opens all available windows in the program, checks all check-boxes, clicks the menus, etc., in a systematic way. The properties of all widgets are written in a form of an XML file.

From the XML file, a text and title was extracted for all recorded widgets. The number of unique strings for each application is in Table 1, column "GUI strings". Examples of these strings include button labels and tooltips, text-area contents, items in combo-boxes and many more. We excluded strings shorter than two characters, as they do not represent realistic search queries for further analysis.

[1] `http://sourceforge.net/projects/freemind/`
[2] `http://sourceforge.net/projects/pdfsam/`
[3] `http://sourceforge.net/projects/weka/`
[4] `http://argouml.tigris.org/`
[5] `http://github.com/AlDanial/cloc`

Table 2 The occurrence counts of whole strings from GUIs in the source code.

	Occurrences of GUI strings in code				
Application	0	1	$[2, 10)$	$[10, 100)$	$[100, \infty)$
ArgoUML	20.5%	10.4%	42.3%	12.1%	14.7%
FreeMind	7.9%	2.8%	60.6%	13.0%	15.6%
PDFsam	13.8%	13.8%	50.8%	4.6%	16.9%
Weka	8.1%	12.0%	14.0%	30.2%	35.6%

Table 3 The occurrence counts of individual words from GUIs in the source code.

	Occurrences of GUI words in code				
Application	0	1	$[2, 10)$	$[10, 100)$	$[100, \infty)$
ArgoUML	4.8%	3.1%	13.6%	29.4%	49.2%
FreeMind	0.7%	0.1%	26.0%	32.9%	40.4%
PDFsam	4.8%	7.7%	4.2%	18.5%	64.9%
Weka	6.6%	0.7%	5.9%	32.5%	54.3%

2.2 Analysis

Some of the strings contain multiple words, or even lines of text. For this reason, we broke them into individual words. We define a word as a sequence of three or more letters. The number of unique words for each application is in the column "GUI words" in Table 1.

For each string contained in the GUI, we searched it in the source code files of the corresponding project. The same process was repeated for individual words.

Regarding the source code, we used tarballs of the same versions as the binaries. The PDFsam tarball contained also automatically generated JavaDoc API documentation, which we removed, since such files should not be included in source archives.

The searching was performed fully automatically, via a script. We decided to perform a case-sensitive search, which should be more precise, especially to locate whole GUI strings. On the other hand, in practice, case-insensitive search is probably the preferred way, as it is often default in IDEs.

3 Quantitative Results

3.1 Occurrence Counts

First, we would like to simulate a situation when a programmer tries to find a whole GUI string in the source code. For each string, we determine a number of occurrences in the project – essentially the number of search results he would get in an IDE. For example, the string "Generate Data" (a button label in the Weka application) has 2 occurrences in the source code of the Weka project. Only text files were searched – this behavior is consistent with the majority of common IDEs which ignore binary files when searching.

In Table 2, we can see what portion of all GUI strings has no occurrence in the source code, exactly one occurrence, from 2 to 10 occurrences, etc.

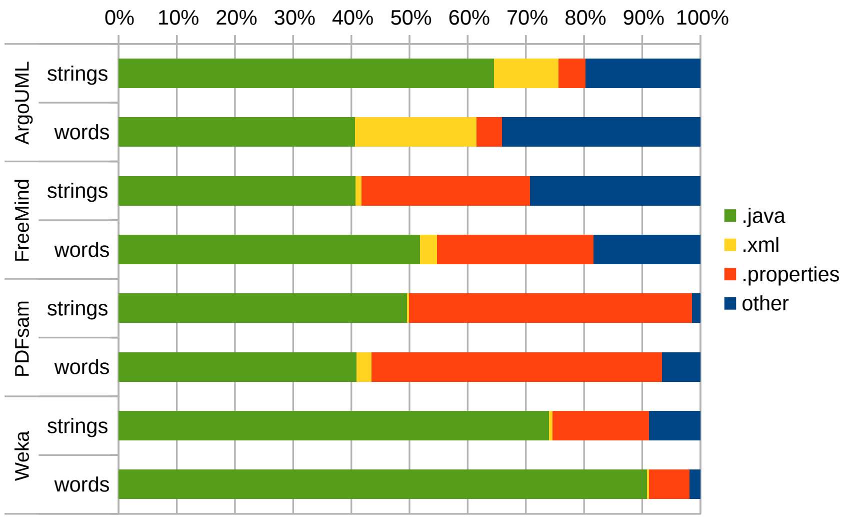

Figure 1 Occurrence counts of strings/words divided by file types.

Similar statistics, but for individual words, are located in Table 3. This represents the situation when the programmer is unhappy about the results and starts searching for smaller parts of the given string – usually words.

An ideal situation arises when a search gives exactly one result. It can (in theory) mean that the programmer found the sole piece of code relevant to the GUI widget. The higher is the number of occurrences, the longer he must sift through search results to find the relevant code. However, a situation when a searched string is not found in the source code is unfavorable, as the developer would have no idea where to start searching for an implementation of the given feature.

3.2 File Types

Ideally, the search results point to Java source files (`*.java`). This way, it is possible to directly find the code of interest. However, the source code of a project usually contains many kinds of files – not just Java source files.

For each project, we took all GUI string (and word) occurrences in all files and divided them by an extension of a file they are located in. In Figure 1, there is a graphical representation of the results.

In ArgoUML, 65% of all occurrences of GUI strings are contained directly in Java source code files. Although the project uses `*.properties` files for internationalization, GUI concepts are often used also as identifiers in Java code. For example, a GUI label "Notation" is present many times in the source code in the form of the class name `Notation`.

The FreeMind project uses "properties" localization files, too.

PDFsam uses a system where each key in a `*.properties` file is the original English string, and the value is the translated one. Therefore, the GUI strings are located both in `*.java` and `*.properties` files.

While Weka also uses `*.properties` files for localization, many GUI strings are also contained in special DSL [7] (domain specific language) files with an extension `.ref`.

4 Qualitative Results

While the numbers gave us some insight about the presence of strings from GUIs in the source code, it is necessary to know exact reasons why some strings were not found at all. Furthermore, we will show on a few samples that the strategy of searching GUI strings in the source code can sometimes be an effective way of finding the relevant code.

4.1 Strings Not Present in Code

Many GUI strings were not found in the source code of the application because they were a part of a universal dialog supplied by the GUI toolkit. For example, color-related terms like "Saturation" were not found in the FreeMind source tree because they are a component of the standard Java Swing color chooser dialog. Examples of such strings in ArgoUML and Weka are "File Name:" – a part of a file selection dialog and "One Side" – a term from the printing dialog.

The string "http://simplyhtml.sf.net/" displayed in FreeMind was located only in a class file inside a third-party JAR archive. Therefore, it is invisible for a standard textual search.

The string "Mode changed to MindMap Mode" was not found in the FreeMind source code as a whole because it was instantiated at runtime from the template "Mode changed to {0}". This forces the developer to find smaller portions of a string until a match appears, as we mentioned earlier.

The label "Show Icons Hierarchically" was not found in the source code of FreeMind because it was written as "Show Icons &Hierarchically" to specify the keyboard shortcut Alt+H.

Examples of strings which are not present in the code as a whole, but their parts can be found there, are help texts, logs and exception stack traces.

Regarding PDFsam, the label "Thumbnail creator:" was not found in the code because the colon was programatically concatenated.

4.2 Strings Present in Code

First, we tried to search for a string which is present exactly once in the FreeMind code: "Change Root Node". It was located in a localization file, as a value of a key named "accessories/plugins/ChangeRootNode.properties_name". Opening the file "accessories/plugins/ChangeRootNode.java" revealed that this Java class is really relevant to the feature.

Next, we searched for a string present 35 times in the code – "Bubble". It represents a node format in the mind-mapping software. This time, the exploration of the results took more time and we required multiple iterations using different search terms, even with some dead ends, until we finally found the relevant source code.

Finally, we searched the string "Export" (a menu item), present 1,988 times in the source code. Just skimming through such a long list is a lengthy activity. Therefore, other strategies are necessary to efficiently find the feature of interest in the code. For example, the programmers can reformulate their search queries, use structured navigation (tools like Call Hierarchy) or debugging techniques [4].

We conclude that in some cases, simple textual searching is a feasible way to find code relevant to a GUI element. Ideally, a GUI string should be located exactly once (or just a few times) in the source code, to allow easy finding of source code relevant to a GUI feature. Furthermore, finding an occurrence in a non-Java file makes it more difficult to find relevant source code than finding it directly in a Java file.

5 Threats to Validity

We will now look at the threats to validity of our study. Construct validity is concerned with the correctness of the measures. External validity discusses whether the findings can be generalized. Reliability denotes whether similar results would be obtained by another researcher replicating the study [13].

5.1 Construct Validity

While the GUI ripper in the GUITAR suite gives good results when scraping the GUI, it is not perfect and it could miss some of the strings visible in the user interface.

During an automated search for whole GUI strings in the code, also long texts like exception stack traces were included. It is not probable that a programmer will actually try to search for a whole stack trace in the code textually, as-is. Instead, he will directly look at some of the methods mentioned in the trace.

As was already mentioned, we performed a case-sensitive search, which has both advantages and disadvantages. In the future, a case-insensitive search should be also performed to better reflect the manual searching behavior of programmers.

5.2 External Validity

All four applications in our study were desktop Java programs, using the Swing GUI widget toolkit. However, common contemporary applications have Web and mobile front-ends. Scraping Web applications could have produced much different results. For example, they often display texts downloaded from external databases. This could be one of the reasons for non-presence of GUI strings in the code of these applications.

Even in the world of Java Swing applications, the selected ones represent just a small sample. However, they are representative of common Java projects, as three of them were included in the standard SF110 [6] benchmark.

5.3 Reliability

The quantitative results were produced chiefly by an automated script. Therefore, the subjectivity of a researcher is eliminated. The strings presented in the qualitative part were selected manually, but we tried to select representative samples.

6 Related Work

6.1 GUI Ripping

To rip GUIs, we used the tool GUI ripper [10] which is a part of the GUITAR [12] suite. Swing UIs are one of the best supported technologies, however, there is a partial support for SWT, Web, and Android. To crawl highly dynamic Web applications, Crawljax [11] could be used.

The DEAL method [1] creates a DSL from a GUI. However, the process is not automated and a user must manually traverse the user interface.

6.2 Feature Location Using GUIs

Of course, finding a string from a UI using IDE's textual search is not a sole option to perform feature location. GUITA [14] allows to take a snapshot of a running GUI widget. The snapshot is associated with a method which was last called on the given widget.

Another approach, UI traces [15], splits a long method trace into smaller ones, each associated with a graphical snapshot of the GUI in the given state.

6.3 Feature Location in General

There exists a large number of feature location techniques – see [5] for a survey. An example of a method utilizing source code comments and identifiers is presented by Marcus et al. [9]. Carvalho et al. [3] use a combination of static and dynamic analysis, specifications and ontologies to map problem domain concepts to source code elements. However, none of these approaches use labels directly from GUIs of running programs.

6.4 Other Studies

Václavík et al. [16] analyzed words used in names of identifiers in the source code of Java EE application servers and web frameworks. They tried to determine what portion of these words are meaningful according to the WordNet database. The more words from the source code of a project are meaningful, the more understandable it should be. We could perform a similar experiment, but use a dictionary built from the GUI of an application instead of WordNet.

7 Conclusion

In this article, a simple study was performed: We scraped all strings contained in a GUI of four open-source Java desktop applications and tried to automatically find them (as a whole and words contained in them) in the static source code.

Regarding the main *hypothesis*: The vast majority of GUI strings were found in the code. However, 11.2% of them were not found at all.

Answering *RQ1*, the GUI strings are often located in Java source code files, `*.properties` localization files, XML and custom DSL files.

To answer *RQ2*, the main reasons of a non-presence of a GUI string in the source code were: a string was a part of a standard dialog, a third-party library, or the string was dynamically generated at runtime.

If we consider a set of GUI words the application's problem domain dictionary, the percentage of GUI words present in the source code can be regarded as a measure of code understandability. For example, if the code contains too few concepts from the GUI, it can be considered obfuscated.

We found out that the approach of simple textual code search of strings displayed in the GUI is useable and it can actually find code relevant to a feature, unless there are too many results, or none at all.

As a future work, there is a potential in creation of a tool which would automatically assign GUI strings to corresponding source code fragments, e.g., using annotations. This way, a simple textual search would be sufficient to quickly find any code related to a given GUI element. Also, we can replicate this study on Web applications, or study logs in addition to GUIs.

References

1 Michaela Bačíková, Jaroslav Porubän, and Dominik Lakatoš. Defining domain language of graphical user interfaces. In José Paulo Leal, Ricardo Rocha, and Alberto Simões, editors, *2nd Symposium on Languages, Applications and Technologies*, volume 29 of *OpenAccess Series in Informatics (OASIcs)*, pages 187–202, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.SLATE.2013.187`.

2 Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE'93, pages 482–498, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press. URL: `http://dl.acm.org/citation.cfm?id=257572.257679`.

3 Nuno Ramos Carvalho, José João Almeida, Pedro Rangel Henriques, and Maria João Varanda Pereira. Conclave: Ontology-driven measurement of semantic relatedness between source code elements and problem domain concepts. In *Computational Science and Its Applications – ICCSA 2014*, pages 116–131. Springer International Publishing, 2014. `doi:10.1007/978-3-319-09153-2_9`.

4 Kostadin Damevski, David Shepherd, and Lori Pollock. A field study of how developers locate features in source code. *Empirical Software Engineering*, 21(2):724–747, 2016. `doi:10.1007/s10664-015-9373-9`.

5 Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013. `doi:10.1002/smr.567`.

6 Gordon Fraser and Andrea Arcuri. A large-scale evaluation of automated unit test generation using EvoSuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2):8:1–8:42, December 2014. `doi:10.1145/2685612`.

7 Tomaž Kosar, Nuno Oliveira, Marjan Mernik, Maria João Varanda Pereira, Matej Črepinšek, Daniela da Cruz, and Pedro Rangel Henriques. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 7(2):247–264, April 2010. `doi:10.2298/CSIS1002247K`.

8 Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Trans. Softw. Eng. Methodol.*, 23(4):31:1–31:37, September 2014. `doi:10.1145/2622669`.

9 Andrian Marcus, Andrey Sergeyev, Václav Rajlich, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223, Nov 2004. `doi:10.1109/WCRE.2004.10`.

10 Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: reverse engineering of graphical user interfaces for testing. In *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, pages 260–269, Nov 2003. `doi:10.1109/WCRE.2003.1287256`.

11 Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling AJAX-based Web applications through dynamic analysis of user interface state changes. *ACM Trans. Web*, 6(1):3:1–3:30, March 2012. `doi:10.1145/2109205.2109208`.

12 Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, 21(1):65–105, 2013. `doi:10.1007/s10515-013-0128-9`.

13 Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009. `doi:10.1007/s10664-008-9102-8`.

14 André L. Santos. GUI-driven code tracing. In *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, pages 111–118, Sept 2012. doi:10.1109/VLHCC.2012.6344495.

15 Andrew Sutherland and Kevin Schneider. UI traces: Supporting the maintenance of interactive software. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 563–566, Sept 2009. doi:10.1109/ICSM.2009.5306389.

16 Peter Václavík, Jaroslav Porubän, and Marek Mezei. Automatic derivation of domain terms and concept location based on the analysis of the identifiers. *Acta Universitatis Sapientiae. Informatica*, 2(1):40–50, 2010. URL: http://www.acta.sapientia.ro/acta-info/C2-1/info21-4.pdf.