Labeling Source Code with Metadata: A Survey and Taxonomy

Matúš Sulír, Jaroslav Porubän
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
Email: {matus.sulir,jaroslav.poruban}@tuke.sk

Abstract—**Source code is a primary artifact where programmers are looking when they try to comprehend a program. However, to improve program comprehension efficiency, tools often associate parts of source code with metadata collected from static and dynamic analysis, communication artifacts and many other sources. In this article, we present a systematic mapping study of approaches and tools labeling source code elements with metadata and presenting them to developers in various forms. We selected 25 from more than 2,000 articles and categorized them. A taxonomy with four dimensions – source, target, presentation and persistence – was formed. Based on the survey results, we also identified interesting future research challenges.**

I. Introduction

DURING their work, developers often encounter situations when they are trying to understand a program by looking at its source code, but the critical information they seek is not present in the code. Consider the following examples.

A small but tricky piece of code worked a few days ago, but now it does not. The programmer must open a web browser, navigate to the version control system (VCS) website and find the relevant commit. It refers to the issue tracking system, which is a separate website. After reading the whole, particularly long issue description and a multitude of related comments, he finally finds the reason of the malfunction.

Another programmer tries to comprehend a rather complicated algorithm. It is difficult to understand it just by looking at the code, so he decides to provide sample input data and debug the program using a built-in debugger of an IDE (Integrated Development Environment). While it is possible to display a value of any variable at any time, the debugger does not present any overview of values of a particular variable over time. Each time a program stops, the programmer must remember a value of interest and compare it with previous values in mind. As the capacity of short-term memory is very limited, the developer soon starts writing notes in a separate document. This, in turn, creates a burden of switching between two separate views (a split-attention effect [1]).

These two – at the first glance unrelated – scenarios have something in common: The information a developer needed was available *sometimes* or *somewhere*. But it was not available in the right place at the right time: in the IDE, and associated with the particular piece of code the developer was looking at.

This work was supported by project KEGA 047TUKE-4/2016 Integrating software processes into the teaching of programming.

Many researchers have realized this problem and provided various approaches, methods and tools to partially solve some of its aspects. However, since authors use a rather large variety of terms to describe them, gaining an overview is difficult. For this reason, we decided to provide a survey of existing approaches and categorize them in a taxonomy.

Our general research questions for this survey are:

- **RQ1**: What various approaches (and tools implementing them) do exist to label parts of source code with additional metadata and present them to a programmer in order to improve program comprehension?
- **RQ2:** How can the approaches be categorized?
- **RQ3:** What observations and challenges can be concluded from the results?

II. Method

We decided to conduct a systematic mapping study, which is a form of a systematic literature review (SLR). In contrast to an SLR, a mapping study has more general research questions [2] and the main goal is to classify research to categories, rather than provide precise quantitative results [3].

A. Search Strategy

Since our view of literature through a notion of "source code labeling" is not very common and the terminology is inconsistent, we decided to try multiple different search strategies and combine their results.

1) Manual Search: First, we performed a manual search among all articles published in 8 journals and 4 conferences, selected by the authors' discretion (partially inspired by a list in [4]). In this first part of the search process, arbitrary two years (2009 and 2012 in our case) were selected, as suggested by [5]. The journals of interest were:

- IEEE Transactions on Software Engineering (TSE),
- ACM Transactions on Software Engineering and Methodology (TOSEM),
- Computer Languages, Systems and Structures (COMLAN),
- Science of Computer Programming (SCP),
- Journal of Systems and Software (JSS),

- Empirical Software Engineering (ESE),
- Information and Software Technology (IST),
- Journal of Software: Evolution and Process (JSEP), formerly known as Journal of Software Maintenance and Evolution (JSME),

and conferences:

- International Conference on Software Engineering (ICSE),
- International Conference on Program Comprehension (ICPC),
- Working Conference on Reverse Engineering (WCRE)
- and International Conference on Software Maintenance (ICSM).

A Scopus[1] query was constructed based on the criteria, the results list was exported as a CSV file and inspected in a spreadsheet processing program. A total of 1546 articles were manually assessed based on titles and abstracts, resulting in a list of 16 relevant articles.

2) Keyword Search: Continuing the methodology of Zhang et al. [5], we inspected the terminology used in articles obtained during the manual search and based on it, we constructed and tried multiple keyword-based search queries. The final Scopus query is as follows:

```
TITLE-ABS-KEY(
  ("source code" OR "program comprehension")
 AND
  ("tagging" OR "enriching" OR "augmenting"
   OR "labeling")
) AND SUBJAREA(COMP)
AND NOT SUBJAREA(bioc OR medi OR envi OR neur)
```

Basically, it searches the specified terms in titles, abstracts and keywords of computer science literature, excluding interdisciplinary research. The search yielded 85 results, of which 6 were newly found relevant ones.

The methodology by Zhang et al. [5] prescribes trying slightly different queries until one of them returns at least 80% of articles from the first (manual) phase. For the query presented above, this number was far below 20%. Broadening the terms caused the count of results to skyrocket. The number of false positives was high, without a significant positive impact on the relevant result count. For this reason, we decided to leave the methodology and continue with other techniques.

3) References Search: We searched for all forward and backward references of 22 articles collected so far. Again, we used Scopus.

Backward references mean all articles cited in the "References" section of particular papers. They are generally older than the article citing them. From 305 results, we considered 14 unique and relevant.

Forward references are articles for which a search engine knows they cite a particular paper. This is useful to find newer articles. Of 137 results, 3 were relevant and not yet found in previous searches.

[1]http://www.scopus.com

4) Other Sources: Five more relevant articles were found recursively in the references of articles found during the phase of references search. Finally, we added three more papers present in the authors' personal bibliography.

B. Inclusion Criteria

During the selection process, a paper was considered relevant if:

- it presented a new approach or tool to associate metadata with pieces of source code,
- the purpose of these metadata was to improve program comprehension
- and a form of presentation of these data to a programmer was described.

Examples of excluded articles are papers describing a labeling algorithm without discussing how to present results to developers, and purely empirical studies comparing existing approaches.

C. Final Article List

From the 47 selected articles, 17 were just descriptions of the same or similar idea in another research phase. Five were considered irrelevant after skimming or reading the full text.

The final article list thus contains 25 articles. For an overview, see Table II. Further details will be provided in section IV.

D. Data Extraction

The full text of 25 relevant articles was read, carefully watching for similar and distinguishing signs regarding source code labeling. Succinct notes about each article were written in a tabular form, gradually forming a taxonomy.

III. TAXONOMY

First, we will introduce our taxonomy, answering **RQ2**. Similar to Dit et al. [6], articles (approaches) were evaluated according to multiple criteria, called dimensions. For each dimension, an article can belong to one or more attributes.

Our taxonomy has four dimensions: source, target, presentation and persistence. For an overview, see Table I. Now we will describe the dimensions and attributes in detail.

A. Source

A "source" dimension denotes where the metadata were originally available before they were assigned to a part of source code. The most problematic source is human mind. In order to obtain information present only in the memory of the programmer, he must manually enter these data into a system for each artifact which should be labeled. The most primitive kind of a label with a "source" of type *human* is a traditional source code comment. The developer writes the label – a natural language text easing program comprehension – above a piece of code. The assignment of the comment to a piece of code is therefore performed by its positioning.

Approaches categorized as *code* analyze the source code of a system without executing it, i.e., using static analysis.

TABLE I
A SOURCE CODE LABELING TAXONOMY.

Dimension	Attribute	Description
Source	*human*	Manually entered information, previously present only in human mind.
	code	Results of static source code analysis.
	runtime	Results of the program execution; dynamic program analysis.
	interaction	Interaction patterns of a single developer in the IDE.
	collaboration	Collaboration artifacts of multiple developers like VCS commits or e-mails.
Target	*folder*	A directory or a package.
	file	A file or a class.
	multi-line	A multi-line part of a file, e.g., a method.
	line	One line in a file, such as a variable declaration.
	line part	A character range, e.g., a method call.
Presentation	*code view*	The editable source code view is augmented with metadata.
	existing view	Other existing views in an IDE (e.g., a package explorer) are augmented.
	separate view	A separate view is created just to present the information of interest.
Persistence	*internal*	The metadata is stored directly in the source code file (e.g., using a comment).
	external	A separate file, database or server is used to store the labels.
	none/unknown	The metadata are only presented to the user, but not stored; or a method of persistence was not mentioned in the article.

Useful metadata can be collected by execution of the program, using some form of dynamic analysis. These approaches are marked as *runtime*. The analyzed program must be buildable (which is often a problem [7]) and automated tests should be available (or the program must be executed manually).

For tools utilizing the *human* source, a programmer must purposefully enter the metadata with a sole intention that they will improve program comprehension. This is expensive on human resources. On the other hand, *interaction* data are collected automatically, possibly without the developer even knowing it (although that would be unethical). Using heuristics, these tools can infer relationships between artifacts from captured keystrokes, mouse actions, or even eye gazes [8].

As software engineering is not an individual activity, collaboration artifacts are formed naturally as the team communicates and collaborates. These artifacts include e-mails, instant messages, forum posts and VCS commit messages. These artifacts are often poorly or nowise connected with the relevant source code. The purpose of *collaboration* approaches to code labeling is to fill this gap.

Many tools use a combination of multiple methods. For example, a static analysis may be used to assign source code artifacts their documentation; then a user must manually confirm or reject the suggested links [9].

B. Target

The purpose of source code labeling is to assign metadata to a particular piece of code. Subject of the “target” dimension is what that “piece of code” means.

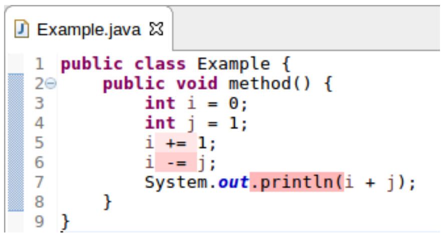
This is quite a problematic question, as there are two separate views: file-based and element-based. Some tools assign metadata to files, lines and character ranges. Other assign them to classes, methods, variables, method calls, etc. These views are often mixed in one tool. Furthermore, it is not clear whether a code bookmark, labeling a line containing just a variable declaration, relates to the line or to the declaration. Therefore, we decided to mix the views in our taxonomy.

The attributes are sorted according to granularity. A *folder* represents a package in some object-oriented languages. A *file* often corresponds to a whole class. Method and function definitions are *multi-line* elements. Elements considered *line* include variable declarations. We decided to consider method calls and variable usages *line parts*.

C. Presentation

Once the metadata were retrieved and associated with a proper source code segment, it should be presented to the developer in order to be useful.

One of the best places to show code-related data is obviously the main, editable source *code view* of an IDE. This is the place which draws the most attention of a programmer, occupies a large screen portion and offers many existing features (e.g., code completion). The code editor can be augmented by various coloring (see Fig. 1), visual overlays (like a box surrounding a piece of code) or images. Harward et al. [10] call them “in situ” visualizations. Syntax highlighting may be considered a common visual augmentation [11]. We decided to regard also gutter/ruler annotations (icons in the left or right code editor margin) as a *code view* presentation.



```
public class Example {
    public void method() {
        int i = 0;
        int j = 1;
        i += 1;
        i -= j;
        System.out.println(i + j);
    }
}
```

Fig. 1. A simple example of *code view* presentation: DepDigger [12] uses background coloring to indicate the understandability of source code parts according to the dep-degree metric.

Except for the source code editor, modern IDEs offer various supplemental views like Package Explorer, Favorites or Class Hierarchy. Plugins can augment *existing views* by additional information. For instance, packages and classes in a tree view may be augmented by colored squares according to a metric [13].

Some tools, despite associating a part of code with additional information, display the association in a *separate* view, or even window. In a better case, clicking a particular widget in this view automatically opens and focuses the code of interest in the source code editor. Otherwise, the user must manually open and find the code according to the displayed element name.

D. Persistence

The association between the code and the label, and sometimes also the label data themselves, can be saved to a permanent storage.

1) Rationale: There are three possible reasons for persistence.

First, in the case of fully automated static *code* and *collaboration*-sourced techniques, the process of retrieving and associating metadata can be resource-intensive. The storage acts as a cache[2].

Second, the *runtime* and *interaction* data are partially a product of human work. Each program execution and IDE interaction can be unique. It is therefore desirable to save at least the data like traces or interaction logs. Once they are persisted, the analysis and association process can be performed every time when necessary.

Third, the persistence is an absolute requirement in the case of *human*-sourced metadata. A tool must not repeatedly ask a programmer to describe code, if exactly the same information had already been entered for the same piece of code, using the same tool.

2) Persistence Methods: The labels can be stored in a source code file itself. The association with the target element is thus implicit through the position of the label in the code. A typical example of *internal* persistence method is a specially formatted comment.

External persistence means labels are stored separately from the source code file. They can be saved in an XML file, or a database management system (DBMS), accessed either locally or through a server.

Finally, the persistence method *none/unknown* denotes the labels are either not stored permanently, or persistence was not mentioned in a given article at all. If a tool produces only reports or exports which cannot be subsequently loaded, it was also incorporated into this category.

[2]Suppose collaboration artifacts are already persisted in external systems.

IV. Approaches and Tools

In Table II, we can see an overview of all reviewed approaches and tools, which answers **RQ1**. Now we will briefly describe each of them. The approaches will be grouped by their primary "source".

A. Human

A concern is a piece of information about a code element, such as a feature it implements or a design decision [32]. ConcernMapper [14] is both an Eclipse plugin and a framework to associate parts of programs with concerns, both through a GUI (graphical user interface) and an API (application programming interface).

The eMoose approach [15] allows tagging of API usage directives, like state restrictions or locking, in JavaDoc comments. Subsequent highlighting of calls to such methods in an IDE improves awareness of programmers, who then spot errors quickly.

Pollicino [16] is a plugin for collective code bookmarks, providing a more feature-rich version of classical source code bookmarks found in the majority of IDEs.

SE-Editor [17] makes it possible to embed web pages in the source code editor of the Eclipse IDE. The web page is embedded using a comment beginning with `/***`, followed by an URL. This might be useful to display code-related diagrams, tutorial videos, etc.

Spotlight [18] is an IDE plugin to tag source code with concerns. Each concern can be assigned a color, which is then displayed in the left gutter (margin) of the source code editor. Thanks to this, a programmer can more easily identify where the given concern is located in the program.

TagSEA [19] integrates waypoints and social tagging into the Eclipse IDE. Programmers note waypoints – places worth marking and sharing – into the source code as comments in the form `//@tag tagname : message`. Hierarchical tags and metadata (author, date) are also supported. Collections of waypoints can be connected into routes to create source code guides.

B. Code

DepDigger [12] visualizes the measure (metric) dep-degree by changing the background color of source code elements – on a scale ranging from white to red – in a code view.

RegViz [20] visually augments a regular expression in-place, without a need for a separate view. For example, groups are underlined and labeled with a group number.

Stacksplorer [21] visualizes method's callers in a column on the left of the code editor view, callees in the right one.

TABLE II
LABELING APPROACHES AND TOOLS.

Article	**Source**					**Target**					**Presentation**			**Persistence**		
	human	code	runtime	interaction	collaboration	folder	file	multi-line	line	line part	code view	existing view	separate view	internal	external	none/unknown
ConcernMapper [14]	■	□	□	□	□	□	□	■	■	□	□	■	■	□	■	□
eMoose [15]	■	□	□	□	□	□	□	■	□	□	■	□	□	■	■	□
Pollicino [16]	■	□	□	□	□	□	□	□	■	□	■	□	■	□	■	□
SE-Editor [17]	■	□	□	□	□	■	■	■	■	□	■	□	□	■	□	□
Spotlight [18]	■	□	□	□	□	□	■	■	■	■	■	□	□	□	■	□
TagSEA [19]	■	□	□	□	□	□	■	■	■	□	■	□	■	■	□	□
DepDigger [12]	□	■	□	□	□	□	□	□	□	■	■	□	■	□	□	■
RegViz [20]	□	■	□	□	□	□	□	□	□	■	■	□	■	□	□	■
Stacksplorer [21]	□	■	□	□	□	□	□	■	□	■	■	□	□	□	□	■
Traceclipse [22]	■	■	□	□	□	■	■	■	□	□	□	□	■	□	■	□
TraceME [9]	■	■	□	□	□	□	■	□	□	□	□	□	■	□	■	□
GUITA [23]	□	□	■	□	□	□	□	□	□	■	■	□	■	□	□	■
Impromptu HUD [11]	□	□	■	□	□	□	□	■	■	□	■	□	□	□	□	■
in situ profiler [1]	□	□	■	□	□	□	□	■	□	■	■	□	□	□	□	■
Senseo [13]	□	□	■	□	□	■	■	■	□	■	■	■	■	□	■	□
sparklines [24]	□	□	■	□	□	□	□	□	■	□	■	□	□	□	□	■
CnP [25]	■	■	□	■	□	□	■	■	■	■	■	□	□	□	■	□
HeatMaps [26]	□	□	□	■	■	□	■	■	□	□	□	■	□	□	■	□
iTrace [8]	■	□	□	■	□	□	■	■	■	□	□	□	■	□	■	□
Deep Intellisense [27]	□	□	□	□	■	□	□	■	□	□	□	□	■	□	■	□
Miler [28]	□	■	□	□	■	□	■	□	□	□	■	■	■	□	■	□
Rationalizer [29]	□	□	□	□	■	□	□	□	■	□	■	□	□	□	■	□
101companies [30]	■	■	□	□	□	□	■	■	■	■	□	□	■	□	■	□
Code Bubbles [31]	■	■	■	□	□	□	■	■	■	■	■	■	■	□	■	□
CoderChrome [10]	□	■	□	■	■	□	□	■	■	■	■	□	□	□	□	■

Graphical overlays may be shown to visually connect the current method definition with the left column and method calls in the source code view to items in the right column.

Traceclipse [22] and TraceME [9] are traceability management recovery tools. They link source artifacts (like documentation) to the target artifacts (usually source code). In the mentioned tools, the linking is performed based on the textual similarity, using IR (information retrieval) methods.

C. Runtime

GUITA [23] annotates GUI-related method calls with GUI snapshots of a selected widget at the time when this method is called. This facilitates the navigation between the dynamic user interface world and the static source code world.

The next three approaches belong to a group of "in situ visualizations" – small graphical elements displayed directly in the source code editor. Impromptu HUD [11] displays a realtime clock-like visual near each scheduled function, informing the programmer when the timing event will fire. An in-situ profiler [1] shows small diagrams with runtime performance information next to method declarations and calls. Code sparklines [24] are small charts depicting values of a particular numeric variable over time.

Senseo [13] gathers and displays dynamic information in static views of an IDE. Information like methods' callers, callees, dynamic (overridden) argument types and return values are shown in a tooltip of a method in the code view. Selected metrics like execution frequencies, object allocation counts or memory consumption can be viewed in gutters/rulers (using heatmaps) and the package explorer (numbers). Dynamic collaborators of packages, classes and methods, and a Calling Context Ring Chart are displayed in a separate view. A controlled experiment demonstrated an improvement of maintenance correctness and speed when using Senseo [13].

D. Interaction

The CnP tool [25] proactively tracks, visualizes and supports editing of code clones in an IDE. Instead of a batch input of source files, the tool captures copy and paste operations in Eclipse.

HeatMaps [26] display artifacts with a background color on a scale from blue to red, according to various numeric values assigned to them. The values are, for example, the recency and frequency of browsing and modification (obtained by instrumenting an IDE), artifact age, and a version count. The blue color means "cold" (e.g, least recently browsed), red "hot". The approach is general and can be potentially applied in many tools and to various views in an IDE.

Interaction-sourced approaches are not limited to traditional mouse and keyboard operations. iTrace [8] analyzes eye gazes (using an eye tracker) in an IDE to infer traceability links between artifacts.

E. Collaboration

Deep Intellisense [27] displays an interleaved list of relevant bugs, commits, e-mails, specifications and other documents for a given code element. Furthermore, a list of related people is shown. The lists are updated each time a user clicks on a code element, but they are displayed in a separate view. Rationalizer [29] integrates similar information directly into the code editor. On the right side of each source code line, it shows three columns: when this line was last modified, who, and why changed it. On the other hand, it processes only data from a VCS and an issue tracker.

Miler [28] is a toolset to retrieve, process and associate e-mail data to source code artifacts. E-mails are assigned to a source code based on textual analysis. Information about e-mails relevant to a class is displayed in the IDE's package explorer and rulers.

F. Mixed Approaches

The following approaches use multiple sources of information, without any of them being dominant.

101companies [30] is a software chrestomathy – a collection of many implementations of one system using various technologies, stored in a repository and linked with metadata. Metadata like an implementation language, dependence on a technology, features, and a highlighting renderer are assigned to files or file fragments using a rule-based DSL (domain specific language). The assignment can be performed according to criteria like a file extension, a regular expression for file content, or even by a separate script [30]. For example, all classes in files called "*.java", ending with "Listener", could be assigned the "observer design pattern" label.

Code Bubbles [31] is a working-set based IDE using fragments called bubbles instead of traditional file-based views. It supports various forms of labeling, ranging from arrows between method calls and definitions, to small images (e.g., a literal "bug") attachable to code bubbles.

CoderChrome [10] provides a generic framework for mapping between a metric and an in-situ augmentation. Examples of such visual augmentations are background colors, glyphs at the beginning or end of lines, and underlining. Metrics can range from categorical to numeric ones, e.g.: a start or end of a block, the last author, the property of having a primitive type, the code age, and a line length.

V. Challenges

To answer **RQ3**, we will now present observations and lessons learned during the categorization and suggestions for future research based on these observations.

A. Filtering Metadata Can Be Necessary

18 of the reviewed tools display various visual annotations and overlays directly in the source code view. Although the idea of tight integration of the source code and metadata is appealing, let us perform a thought experiment: Imagine these 18 approaches are implemented as plugins of one IDE, all installed and enabled at the same time. The amount of metadata could overwhelm the developer, causing more harm than good. It would be interesting to substantiate the thought experiment and perform a real case study with actual plugins. However, since many plugins are unstable academic projects and they are implemented for various IDEs, a reimplementation of many of them would be required.

Suppose the study showed us the amount of metadata is overwhelming. One option to tackle this is to turn the plugins on and off manually each time the developer needs a particular kind of information. This interrupts the programmer and causes additional mental overhead. Therefore, we can expect the majority of tools would not be used at all.

Ideally, the relevant metadata would be shown only when it is necessary. Each tool should formally provide a list of source code characteristics, task kinds and other relevance indicators – in which situations and contexts is this tool useful. The IDE would then calculate numerical relevance based on these characteristics and display only metadata with a relevance higher than a given threshold.

B. Evolution Needs to Be Taken into Account

Both the source code and metadata evolve over time. Each "persistence" type has its advantages and disadvantages regarding evolution.

If a piece of metadata is stored using *internal* persistence, it is pushed to a VCS each time it is updated. This can cause unnecessary overhead during collaboration – e.g., during merging. On the other hand, if the source code itself changes (and the metadata remains valid), there is no referencing problem.

When using *external* storage, the target code part must be explicitly referenced. The most primitive example is referencing by a line number. However, as the source code file content changes with each revision, the original line number may no longer contain the same element. Reiss [33] compared various methods for tracking source code locations, e.g., storing an exact line content, a context of a few lines around the line, AST matching, a diff-based approach, and their combinations.

While some combinations of methods achieve correctness above 97%, none of them is 100% accurate. Furthermore, the performance (space to store the reference and time to compute it) must be taken into account.

Using neither internal nor external storage solves the problem with evolution. However, it is generally advisable only for the *code* source (static analysis). For example, not storing metadata from dynamic analysis causes data loss as soon as the tool is closed.

In the reviewed articles, the effects of evolution, especially what happens if the source code itself changes, were rarely discussed. Empirical evaluations of tools taking evolution into account are necessary.

C. What If Source Code Was Updated by Tools?

One particularly interesting observation can be made by looking at patterns in Table II. All approaches using *internal* persistence use purely *human* source of information. This means that automated tools do not write metadata to the source code files themselves.

We investigated this matter and described preliminary approaches which write the results of dynamic and static analysis directly into the source code, e.g., in a form of Java annotations [34], [35]. Another example is our recent prototype of a tool writing automatically generated Javadoc comments directly into source code files [36]. The mentioned approaches have advantages and drawbacks already mentioned in section V-B.

An interesting way to utilize *internal* persistence for non-human sources would be for IDE-independent program comprehension tools. A tool would temporarily annotate the source code with metadata, using annotations or comments. Therefore, they will be viewable with any IDE or even a simple text editor. Then, just before committing the modified source code to a VCS (or even sooner, when the developer would not need the metadata for comprehension anymore), the tool would remove the metadata, so the source code would remain clean. A disadvantage of this approach is its limitation only to textual metadata.

VI. Threats to Validity

The set of articles included in this study is by no means complete. Nevertheless, due to a large number of papers pertaining to a research area, collecting all related articles is often unrealistic and it is just important for the selected subset to be representative with respect to the research field [2]. We did not attempt to collect all available evidence – our main goals were to present at least a portion of approaches and tools using a preliminary mapping study, construct the taxonomy, and portray future challenges.

One could argue that the whole search process relies on one search resource – Scopus. However, during backward references search, also papers not indexed by Scopus were returned (secondary documents in their terminology).

Although the oldest included articles are from 2005, the selection was not artificially limited to any specific date.

Article selection and data extraction were performed by a sole researcher, which could produce biased results based on subjective decisions. Brereton et al. [37] suggest either independent extraction by at least two researchers and then a comparison of results, or checking the data afterward. In case of the lack of resources, a random sample of data may be cross-checked [3].

This paper is focused on tools presented in academic articles. However, there exist many industrial tools not described in papers, which could be also worthwhile to describe.

VII. Related Work

The research area of feature location partially overlaps with source code labeling – features can be considered one of possible source code labels. Dit et al. [6] reviewed 89 feature location techniques and classified them into a taxonomy. Our "source" dimension is similar to their "type of analysis"; and "target" to "output". On the other hand, we were more concerned about the presentation and persistence.

The *runtime* source in our taxonomy indicates a use of various dynamic analysis approaches. Cornelissen et al. [38] reviewed 176 articles related to dynamic source code analysis. However, labeling parts of source code with obtained information was out of the scope of the mentioned survey.

The goal of traceability research [39] is to allow following links among various forms of software artifacts. Often the target artifact is source code, just as in the case of source code labeling. Two traceability tools, Traceclipse [22] and TraceME [9], were included in this mapping study.

In software engineering, recommendation systems [40] suggest relevant information based on the developer's context. The context can be implicit (e.g., IDE history), explicit (a query) or a combination of them. Robillard et al. [41] recognized multiple design dimensions of recommendation systems. Their "data" dimension is similar to our "source". They also recognized the "presentation" dimension, but in their case, it has values *batch* and *inline*.

Software artifact summarization [42] aims to create shorter descriptions from longer pieces of code, bug reports, mailing lists and discussions. In this sense, summaries can be considered source code labels obtained from the *code* and *collaboration* sources.

Source code labeling with the *collaboration* source often seeks to improve workspace awareness [43] – knowledge of the tasks and artifacts of others in a distributed software development team.

VIII. Conclusion and Future Work

In this systematic mapping study, 2,091 articles were assessed in total (including duplicates), from which 25 unique and relevant articles were selected and briefly described. A taxonomy of source code labeling containing four dimensions was created by the analysis of the articles.

Thanks to this survey, researchers can both get a quick overview of many source code labeling approaches, and find interesting future research directions by analyzing the gaps.

IDE developers can use it as an inspiration about which features to implement in their product. Practitioners struggling to analyze existing large codebases may find information about available tools and approaches here.

We identified three main challenges in the area of source code labeling. First, it will be necessary to filter the metadata displayed directly in the source code view only to the most relevant information – to prevent visual clutter. Second, the evolution of both source code and metadata need to be taken into account when designing tools. Finally, writing the metadata directly into source code files could be promising if implemented properly.

Note that some of the approaches described here are already implemented in industrial IDEs. Furthermore, there exist some features of commercial IDEs not described here. It would be interesting to compare code labeling features of common IDEs and academic tools.

This paper provided only a high-level qualitative overview of the topic. More in-depth analysis, like the quantification of the amount of necessary human work for each approach, is left as future work. We can also introduce more taxonomy dimensions and attributes.

References

[1] F. Beck, O. Moseler, S. Diehl, and G. Rey, "In situ understanding of performance bottlenecks through visually augmented code," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, May 2013. doi: 10.1109/ICPC.2013.6613834 pp. 63–72.

[2] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Information and Software Technology*, vol. 64, p. to appear, 2015. doi: 10.1016/j.infsof.2015.03.007

[3] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Keele University and Durham University Joint Report, Technical Report EBSE-2007-01, Jul. 2007. [Online]. Available: http://community.dur.ac.uk/ebse/guidelines.php

[4] D. Sjøberg, J. Hannay, O. Hansen, V. Kampenes, A. Karahasanović, N.-K. Liborg, and A. Rekdal, "A survey of controlled experiments in software engineering," *Software Engineering, IEEE Transactions on*, vol. 31, no. 9, pp. 733–753, Sep. 2005. doi: 10.1109/TSE.2005.97

[5] H. Zhang, M. A. Babar, and P. Tell, "Identifying relevant studies in software engineering," *Information and Software Technology*, vol. 53, no. 6, pp. 625–637, 2011. doi: 10.1016/j.infsof.2010.12.010 Special Section: Best papers from the APSECBest papers from the APSEC.

[6] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013. doi: 10.1002/smr.567

[7] M. Sulír and J. Porubän, "A quantitative study of Java software buildability," in *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU 2016. New York, NY, USA: ACM, 2016. doi: 10.1145/3001878.3001882 pp. 17–25.

[8] B. Walters, M. Falcone, A. Shibble, and B. Sharif, "Towards an eye-tracking enabled IDE for software traceability tasks," in *Traceability in Emerging Forms of Software Engineering (TEFSE), 2013 International Workshop on*, May 2013. doi: 10.1109/TEFSE.2013.6620154 pp. 51–54.

[9] G. Bavota, L. Colangelo, A. De Lucia, S. Fusco, R. Oliveto, and A. Panichella, "TraceME: Traceability management in eclipse," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, Sep. 2012. doi: 10.1109/ICSM.2012.6405343 pp. 642–645.

[10] M. Harward, W. Irwin, and N. Churcher, "In situ software visualisation," in *Software Engineering Conference (ASWEC), 2010 21st Australian*, Apr. 2010. doi: 10.1109/ASWEC.2010.18 pp. 171–180.

[11] B. Swift, A. Sorensen, H. Gardner, and J. Hosking, "Visual code annotations for cyberphysical programming," in *Live Programming (LIVE), 2013 1st International Workshop on*, May 2013. doi: 10.1109/LIVE.2013.6617345 pp. 27–30.

[12] D. Beyer and A. Fararooy, "DepDigger: A tool for detecting complex low-level dependencies," in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, Jun. 2010. doi: 10.1109/ICPC.2010.52 pp. 40–41.

[13] D. Röthlisberger, M. Härry, W. Binder, P. Moret, D. Ansaloni, A. Villazón, and O. Nierstrasz, "Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks," *Software Engineering, IEEE Transactions on*, vol. 38, no. 3, pp. 579–591, May 2012. doi: 10.1109/TSE.2011.42

[14] M. P. Robillard and F. Weigand-Warr, "ConcernMapper: Simple view-based separation of scattered concerns," in *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, ser. eclipse '05. New York, NY, USA: ACM, 2005. doi: 10.1145/1117696.1117710 pp. 65–69.

[15] U. Dekel and J. Herbsleb, "Improving API documentation usability with knowledge pushing," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, May 2009. doi: 10.1109/ICSE.2009.5070532 pp. 320–330.

[16] A. Guzzi, L. Hattori, M. Lanza, M. Pinzger, and A. van Deursen, "Collective code bookmarks for program comprehension," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, Jun. 2011. doi: 10.1109/ICPC.2011.19 pp. 101–110.

[17] P. Schugerl, J. Rilling, and P. Charland, "Beyond generated software documentation – a Web 2.0 perspective," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, Sep. 2009. doi: 10.1109/ICSM.2009.5306385 pp. 547–550.

[18] M. Revelle, T. Broadbent, and D. Coppit, "Understanding concerns in software: insights gained from two case studies," in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, May 2005. doi: 10.1109/WPC.2005.43 pp. 23–32.

[19] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby, "Shared waypoints and social tagging to support collaboration in software development," in *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, ser. CSCW '06. New York, NY, USA: ACM, 2006. doi: 10.1145/1180875.1180906 pp. 195–198.

[20] F. Beck, S. Gulan, B. Biegel, S. Baltes, and D. Weiskopf, "RegViz: Visual debugging of regular expressions," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014. doi: 10.1145/2591062.2591111 pp. 504–507.

[21] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers, "Stacksplorer: Call graph navigation helps increasing code maintenance efficiency," in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '11. New York, NY, USA: ACM, 2011. doi: 10.1145/2047196.2047225 pp. 217–224.

[22] S. Klock, M. Gethers, B. Dit, and D. Poshyvanyk, "Traceclipse: An Eclipse plug-in for traceability link recovery and management," in *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, ser. TEFSE '11. New York, NY, USA: ACM, 2011. doi: 10.1145/1987856.1987862 pp. 24–30.

[23] A. L. Santos, "GUI-driven code tracing," in *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, Sep. 2012. doi: 10.1109/VLHCC.2012.6344495 pp. 111–118.

[24] F. Beck, F. Hollerich, S. Diehl, and D. Weiskopf, "Visual monitoring of numeric variables embedded in source code," in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, Sep. 2013. doi: 10.1109/VISSOFT.2013.6650545 pp. 1–4.

[25] D. Hou, P. Jablonski, and F. Jacob, "CnP: Towards an environment for the proactive management of copy-and-paste programming," in *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, May 2009. doi: 10.1109/ICPC.2009.5090049 pp. 238–242.

[26] D. Röthlisberger, O. Nierstrasz, S. Ducasse, D. Pollet, and R. Robbes, "Supporting task-oriented navigation in IDEs with configurable HeatMaps," in *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, May 2009. doi: 10.1109/ICPC.2009.5090052 pp. 253–257.

[27] R. Holmes and A. Begel, "Deep Intellisense: A tool for rehydrating evaporated information," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008. doi: 10.1145/1370750.1370755 pp. 23–26.

[28] A. Bacchelli, M. Lanza, and M. D'Ambros, "Miler: A toolset for exploring email data," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011. doi: 10.1145/1985793.1985984 pp. 1025–1027.

[29] A. W. Bradley and G. C. Murphy, "Supporting software history exploration," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011. doi: 10.1145/1985441.1985469 pp. 193–202.

[30] J.-M. Favre, R. Lämmel, M. Leinberger, T. Schmorleiz, and A. Varanovich, "Linking documentation and source code in a software chrestomathy," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, Oct. 2012. doi: 10.1109/WCRE.2012.43 pp. 335–344.

[31] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., "Code Bubbles: Rethinking the user interface paradigm of integrated development environments," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010. doi: 10.1145/1806799.1806866 pp. 455–464.

[32] M. Sulír, M. Nosáľ, and J. Porubän, "Recording concerns in source code using annotations," *Computer Languages, Systems & Structures*, vol. 46, pp. 44–65, Nov. 2016. doi: 10.1016/j.cl.2016.07.003

[33] S. P. Reiss, "Tracking source locations," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008. doi: 10.1145/1368088.1368091 pp. 11–20.

[34] M. Sulír and J. Porubän, "Semi-automatic concern annotation using differential code coverage," in *2015 IEEE 13th International Scientific Conference on Informatics*, Nov. 2015. doi: 10.1109/Informatics.2015.7377843 pp. 258–262.

[35] M. Sulír and J. Porubän, "Exposing runtime information through source code annotations," *Acta of Electrotechnica et Informatica*, vol. 17, no. 1, pp. 3–9, Apr. 2017.

[36] M. Sulír and J. Porubän, "Generating method documentation using concrete values from executions," in *6th Symposium on Languages, Applications and Technologies (SLATE'17)*, 2017.

[37] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," *Journal of Systems and Software*, vol. 80, no. 4, pp. 571–583, 2007. doi: j.jss.2006.07.009

[38] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *Software Engineering, IEEE Transactions on*, vol. 35, no. 5, pp. 684–702, Sep. 2009. doi: 10.1109/TSE.2009.28

[39] S. Winkler and J. Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *Softw. Syst. Model.*, vol. 9, no. 4, pp. 529–565, Sep. 2010. doi: 10.1007/s10270-009-0145-0

[40] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, Eds., *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.

[41] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, vol. 27, no. 4, pp. 80–86, Jul. 2010. doi: 10.1109/MS.2009.161

[42] N. Nazar, Y. Hu, and H. Jiang, "Summarizing software artifacts: A literature review," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 883–909, 2016. doi: 10.1007/s11390-016-1671-1

[43] I. Steinmacher, A. P. Chaves, and M. A. Gerosa, "Awareness support in distributed software development: A systematic review and mapping of the literature," *Comput. Supported Coop. Work*, vol. 22, no. 2-3, pp. 113–158, Apr. 2013. doi: 10.1007/s10606-012-9164-4