Metadata-based Code Example Embedding

Mock skills
Mock skills
mock@skills.com

BP Valid
Bp Valid
bp@valid.com

Extra Terestrial
Extra Terestrial
extra@terestrial.com

ABSTRACT

In practice, developers usually seek different ways to save time and effort. Thus, they use different tools (such as search engines, issue tracking, or Q&A sites) to collaborate and find code examples that can meet their specific needs. However, such tools only support the traditional *find-alter-embed* approach of code examples while ignoring the origin and location of these sources. Such information can be very useful to assist software development tasks such as bug-fixing, teamwork, and knowledge transfer, through direct notification of critical changes made to the code example, or access to the original source including its discussions, issues, and bug reports.

In this paper, we propose a new approach that consists of collecting meta information about a code example to automatically track critical changes to it and its origin and provide feedback to both developers and the online community. We report on our vision, approach and challenges, and draft a software architecture to implement our research idea.

KEYWORDS

Software process; Code Example Embedding; Mining Software Repositories; Collaborative Software Development; Development Environment

ACM Reference format:
Mock skills, BP Valid, and Extra Terestrial. 2017. Metadata-based Code Example Embedding. In *Proceedings of 3rd International Workshop on Software Analytics, Paderborn, Germany, July 5–7, 2017 (SWAN'17),* 4 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Code reuse is a form of knowledge reuse in software development [9], e.g., reuse of an existing code example to derive code from one's own project. Barzilay proposes Example Embedding (EE) as a new activity to support code reuse with the following three main activities: *finding*, *adapting*, and *embedding* of examples [2].

Many researchers have investigated different techniques, such as, search engines [1, 12], API mining [28, 30, 34], or source code recommendation [17, 18, 33] to support EE and other software development activities [4, 7, 8, 16]. Techniques for EE usually refer to meta information about the code context (IDE-settings, package-, class-, method-name), or even social media (votings, users experiences, discussions) to find, rank, and evaluate code examples. When integrated in the development environment (IDE), these techniques can even help the developer stay focused on his task [3, 18]. However, once the example is introduced into the target software, for instance, through copy-pasting (or retyping) the code text, the link with the original source is often lost. This leads to the following drawbacks:

- The developer might miss critical changes [1] to the original source, for instance if the example was taken from an external resource (such as a code repository or a Q&A site).
- He might create unintentional clones [5] (known as bad smell [20]) along with other team members which might be harmful [11] and might lead to severe bug propagation [14]. Even if changes are shared using a revision control system (RCS), a possible daily can lead to new clones.
- He might also need to switch between different contexts to find alternative solutions or share his knowledge with team members or the crowd.

The scenario in Figure 1 highlights the above shortcomings by describing the traditional *adapt-test-commit* approach that was first presented here [14]. In this scenario, a code example from a web resource (1) that contains a bug is first shared to the team's revision repository (2). After the bug is detected, it is adapted and tested (3), then the fixed code is committed (checked-in) into the revision repository through the same channel (4). Although this approach is very common among developers, it requires each occurrence to be first analyzed, fixed, tested, and then committed. To some extent, the damage might have already been done by then. Other team members might have copied (or worse adapted) the buggy code several times, or started working on other tasks by then, and would have to remember each single copy in order to apply the fix [29].

We believe that keeping the link to the original source can help tackle these shortcomings. First, by keeping the original information and reducing the risk of fault propagation through lack of communication. Second, by providing useful feedback about possible defects and/or optimizations to both developers and the online community.

Thus, we propose a new approach to EE that extends the traditional *find-alter-embed* with *tracking* of code examples. It consists of first, maintaining relevant code metadata about the example in order to track how it evolves over time. For example, information about the origin and/or location of the example (like URL, commit information, package-, class-name etc). Or, fine-grained metadata [2] which can store information about potential clone-clonee relationships for some piece of code. We propose to use such meta information to automatically *track* changes to the example and its

SWAN'17, Paderborn, Germany

DOI: 10.475/123_4

[1] Critical changes are "better alternatives", "useful discussions to example", or "reports about issues and bugs", see section 2
[2] Fine-grained code metadata is metadata that is attached to pieces of code, e.g., a code block, some lines or only a few characters [19].

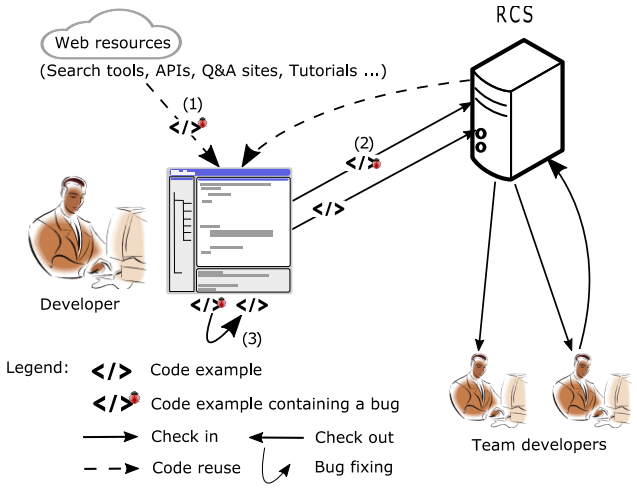

Figure 1: Traditional "adapt-test-commit" approach

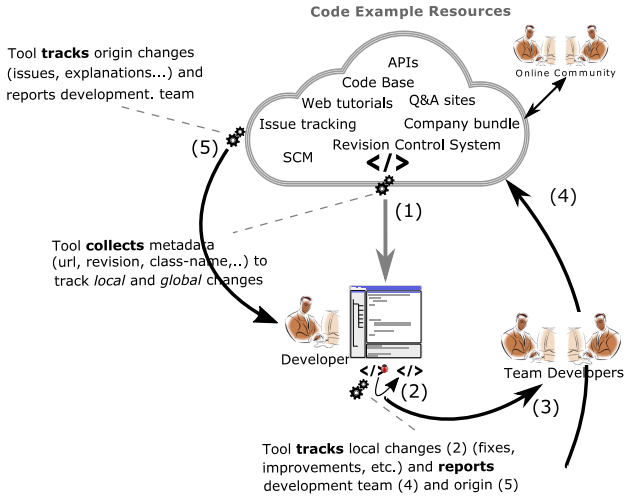

Figure 2: Our approach of Metadata based Example Embedding

original source, and to share knowledge between team developers and the online community.

Section 2 describes our vision for metadata example embedding and tracking, presents a list of challenges to it, and proposes a software architecture. Next, we review related work (Section 3). Finally, we conclude (Section 4).

2 RESEARCH DIRECTION

To realize our vision, we must address a set of challenges. This section presents these challenges and proposes a software architecture to our vision.

2.1 Vision and challenges

To be able to track changes to a code example and provide feedback, our tool (as shown in Figure 2) needs to collect any relevant information about the origin or location of the example before it is introduced into the target source code (1). For example, the URL of the page where the example was taken from, the position of the code snippet in the web page (or discussion thread), commit information (like author, revision number, commit date, issue or bug number), or the location of the example in the target source like project-, package-, or class-name. Also, fine-grained code metadata can be collected, like clone-clonee relationships, or merge edit and refactoring operations [6].

Our tool employs techniques such as search engines, or Editor/Browser plugins to gather the metadata directly from the original source of the piece of code. Then, it tracks changes to the embedded example (2) like adding, altering, or deleting code lines or comments, renaming of variables, etc. The collected information, when considered critical, is then reported to the team developers and/or the online community (3), (4). Our tool also notifies the developer about new events which are occuring in the original source (5). For instance, events in a Q&A site would be new comments to the original code example, or a higher voting of another code example. Other events might be new reports about bug fixes or updates provided by an issue tracker.

All this raises a list of research challenges that have to be addressed: **What metadata is relevant and should be collected? [R1] What is a critical code change? [R2] How to efficiently communicate changes and share knowledge between developers and the online community? [R3]**

We will address these questions in the next section and provide a basic, conceptual architecture for our vision.

2.2 Architecture

In order to tackle the problems described above we propose to implement a client and a server component. Figure 3 shows the basic, conceptual architecture.

The client component needs to be as seamlessly integrable as possible into the development environment to reduce unnecessary cognitive overhead for the developer (see Section 1). Technically speaking, this can be achieved by using plug-in infrastructures as they are provided by IDEs or browsers. The server component is supposed to handle all tracking, collecting and analyzing tasks which are required to address our research questions. In the following section the subcomponents of the client (Section 2.2.1) and server (Section 2.2.2) are explained in detail.

2.2.1 Client component. The primary task of the client component is the collection of information about the embedded code example–the metadata. Therefore, a '**Metadata Collector**' subcomponent collects all accessible information which can be obtained from the origin of the code example (e. g. URL, position of the code snippet in the website, or revision information etc.). Technically, in an IDE this can be achieved by using the internal web browser for searching the code example and, thus, having access to the website itself.

The collected metadata then is available to client component and, furthermore, is sent to the server component where it is stored and will be analyzed further (like the determination of relevance and

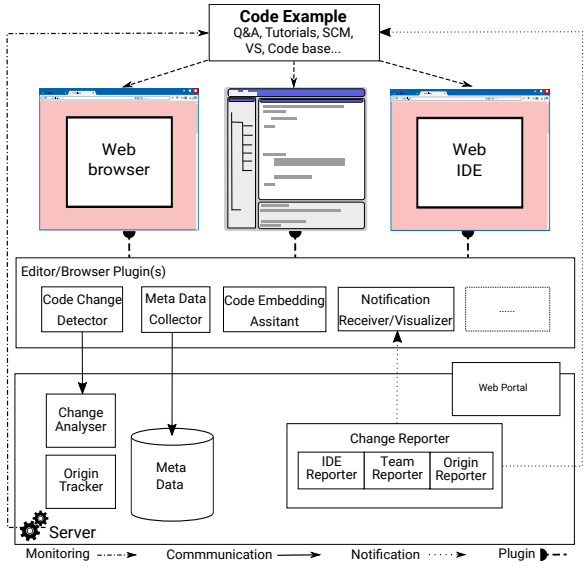

Figure 3: Proposed architecture for Metadata based Example Embedding

criticality) [R1]. In the client the metadata are shown as visual annotations (e. g. as markers or quick tips) on the portions of the source code [R3] which has been embedded by the developers. Besides the annotations also notifications about different changes on the origin of the example are presented to the developer [R3]. These notifications are generated by the server and are received and visualized by the client subcomponent '**Notification Receiver/Visualizer**'.

Next to the described subcomponents a variety of further functionality can be integrated into the client component which can enhance the handling of code examples in general. Figure 3 shows two further examples: the '**Code Change Detector**' and the '**Code Embedding Assistant**'. The latter one eases the embedding of the code example for instance by modifying the source code in order to fit the target domain of the source code. Those features basically are build on principles as presented in [10, 21, 31]. The 'Code Change Detector' detects all changes which are made to the embedded code example (like adding, altering, or removing a line or comments) and notifies the server about these changes. This information is used to track the evolution of the source code in general and the code example in particular. Here, again existing approaches as presented in [6, 15, 22] can be integrated in subcomponents of the clients.

2.2.2 Server component. The server component collects all information that is provided by the client (like metadata or change events). The collected information is stored and analyzed by specific subcomponents in order to determine whether it is a relevant [R1] or critical information/change [R2]. For instance, if metadata about a new code example is received, the data is stored and then the server invokes the '**Origin Tracker**' for further analysis of the origin of the code example. For a Q&A site further metadata like the thread history or the rating of the particular code example (*relevant information*) is collected. In the end the server analyzes all collected metadata and tries to find further insights about the code example which may result in further metadata [R1]. The same is true for the 'Change Analyzer'. It analyzes all changes received by the client (e. g. clone analysis) in order to collect or compute new metadata about the code example [R1,R2].

Besides collecting and creating metadata, the server is supposed to inform developer, development team and online community about changes of and new information about the code example. Therefore, the subcomponent '**Change Reporter**' is creating notifications for events which are occurring. Example events for a Q&A site would be a higher voting of another code example, a new comment or a bug-fix in the origin of the embedded code example (*critical change*) [R2]. The main challenge regarding the probably high number of generated events is the proper filtering and aggregation of relevant events to a particular developer or development team including the selection of appropriate receivers [R3].

3 STATE OF THE ART

Code tracking denotes the process of tracing how a piece of code evolves over time, moves between files inside a repository, and crosses repository boundaries. It is the prerequisite for maintaining code metadata, i.e., data about the piece of code [19]. This fine-grained metadata, in turn, can store the information about potential clone-clonee relationships for every piece of code, and help us realize our vision.

Software repositories are databases that manage the artifacts that software projects generate, for instance, revision control system (RCS), archived communications, and issue tracking systems. Mining all kinds of repositories that can contain pieces of code is relevant to our work. Revision repositories are primarily important for this work as they contain a project's most fundamental artifacts, the source code, and manage it in the dimensions of space (directories and files) and time (evolution from revision to revision) [27]. Yet their view is incomplete, as they only get to see snapshots at discrete moments in time when code is checked-in (cf. [19]). This is a problem for code tracking, as a piece of code might be copied and modified before it is checked-in to a repository, complicating tracing of true origins, and thereby assignment of correct clone-clonee relationship metadata.

Hence, the two major technical problems are (a) *editing analysis* to determine whether and how a certain piece of code is modified to maintain existing clone-clonee metadata, and (b) *clone detection* to establish a clone-clonee relationship in the first place, and then maintain existing metadata when code is moved. Proposals like [6, 15, 22] can improve the situation with respect to tracking. However, such technologies have not yet penetrated the software repository market in the necessary breadth — meaning that every repository that serves as a source for code examples supports tracking — and possibly might never do so.

Typical approaches to *editing analysis* rely on various algorithms of string editing, e.g., the Levenshtein distance. Navarro [13] provides a comprehensive introduction to string editing. But also tree editing [24] applied to the abstract syntax trees of code, or refactoring identification [32] may be necessary. *Clone detection* involves determining what code is suspected as being cloned, to find potential clonees, and to decide whether a suspected clone-clonee relationship actually exists. Of course, this has all the limitations of recall (missing clones) and precision (wrongly assuming a clone

relationship where there is none; cf. [23]). A much more detailed overview of clone detection than we can provide here can be found in Roy et al. [25]. A basic method of *maintaining fine-grained code metadata* while code is edited and moved is presented in [19] but may need adaption to our purposes.

Traditional clone detection also rarely considers sources other than the project's revision repository as originating source of clones. Software product lines already broaden the scope by leaving the same-repository domain (cf. [26]). However, with respect to our work, we must cross all boundaries that also a developer might cross in his search for examples, and find cloned code in other types of tools (e.g., issue trackers, blogs, wikis) and tools outside the respective social context (e.g., outside the one's own company, books, or the internet). Particularly relevant here is the field of code example recommendation. For example, social media have gained considerable importance in recent years as source of code examples [33]. While we do not want to recommend code examples in our work, we must acknowledge that such tools and platforms are accessed by developers. Hence, we must be able to access, analyze and monitor these sources. For doing so, many other technologies become important, e.g., web crawling or data management.

4 CONCLUSION

Until now, researches in Example Embedding have only focused on the traditional *find-alter-embed* [2] approach of code examples, and have almost ignored valuable meta information such as the origin or location of these sources. We propose to maintain and use such information to automatically *track* critical changes to code examples and their origins and provide useful feedback to the developer, team developers, and the online community. In this paper, we have presented a first architecture to implement this approach and provided a series of research challenges that need to be addressed. In the next steps, we will use expert interviews to validate our proposed architecture on conceptual and technical levels.

REFERENCES

[1] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 681–682.

[2] Ohad Barzilay, Orit Hazzan, and Amiram Yehudai. 2009. Characterizing Example Embedding as a software activity. In *ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. IEEE Computer Society, 5–8.

[3] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 513–522.

[4] Fuxiang Chen and Sunghun Kim. 2015. Crowd debugging. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 320–332.

[5] James R Cordy, Thomas R. Dean, and Nikita Synytskyy. 2004. Practical language-independent detection of near-miss clones. In *Conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1–12.

[6] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. 2007. Refactoring-Aware Configuration Management for Object-Oriented Programs. In *International Conference on Software Engineering*. IEEE Computer Society, 427–436.

[7] Max Goldman, Greg Little, and Robert C Miller. 2011. Real-time collaborative coding in a web IDE. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 155–164.

[8] Anja Guzzi, Alberto Bacchelli, Yann Riche, and Arie van Deursen. 2015. Supporting developers' coordination in the ide. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*. ACM, 518–532.

[9] Stefan Haefliger, Georg Von Krogh, and Sebastian Spaeth. 2008. Code reuse in open source software. *Management Science* 54, 1 (2008), 180–193.

[10] Patricia Jablonski and Daqing Hou. 2007. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *OOPSLA workshop on eclipse technology eXchange*. ACM, 16–20.

[11] Cory Kapser and Michael W Godfrey. 2006. "Cloning considered harmful" considered harmful. In *Working Conference on Reverse Engineering*. IEEE, 19–28.

[12] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. 2010. Towards an Intelligent Code Search Engine.. In *AAAI*.

[13] Gonzalo Navarro. 2001. A Guided Tour to Approximate String Matching. *Comput. Surveys* 33, 1 (2001), 31–88.

[14] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H Pham, Jafar Al-Kofahi, and Tien N Nguyen. 2012. Clone management for evolving software. *IEEE Transactions on Software Engineering* 38, 5 (2012), 1008–1026.

[15] Takayuki Omori and Katsuhisa Maruyama. 2011. A Software Development Environment Maintaining Fine-grained Code Metadata by Using Editing Operations. *IASTED Intl. Conference on Software Engineering* (2011), 144–151.

[16] Raphael Pham, Yauheni Stoliar, and Kurt Schneider. 2015. Automatically recommending test code examples to inexperienced developers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 890–893.

[17] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. 2013. Leveraging crowd knowledge for software comprehension and development. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 57–66.

[18] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 102–111.

[19] Christian R. Prause. 2009. Maintaining fine-grained code metadata regardless of moving, copying and merging. In *Intl. Conf. on Source Code Analysis and Manipulation*. IEEE, 109–118.

[20] Foyzur Rahman, Christian Bird, and Premkumar Devanbu. 2012. Clones: What is that smell? *Empirical Software Engineering* 17, 4-5 (2012), 503–530.

[21] Steven P Reiss. 2009. Semantics-based code search. In *International Conference on Software Engineering*. IEEE CS, 243–253.

[22] Romain Robbes. 2007. Mining a Change-Based Software Repository. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*. IEEE Computer Society Washington, DC, USA.

[23] Romain Robbes and Michele Lanza. 2005. Versioning Systems for Evolution Research. *Principles of Software Evolution, International Workshop on* (2005), 155–164.

[24] Sebastian Rönnau and Uwe Borghoff. 2012. XCC: change control of XML documents. *Computer Science - Research and Development* 27 (2012), 95–111. Issue 2. DOI: http://dx.doi.org/10.1007/s00450-010-0140-2

[25] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470 – 495. DOI: http://dx.doi.org/10.1016/j.scico.2009.02.007

[26] Sandro Schulze. *Analysis and Removal of Code Clones in Software Product Lines*. Ph.D. Dissertation.

[27] Diomidis Spinellis. 2003. *Code Reading – The Open Source Perspective*. Addison Wesley.

[28] Jeffrey Stylos and Brad A Myers. 2006. Mica: A web-search tool for finding api components and examples. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*. IEEE, 195–202.

[29] Stefan Wagner. 2015. Continuous and Focused Developer Feedback on Software Quality (CoFoDeF). *Research Ideas and Outcomes* 1 (2015), e7576.

[30] Lijie Wang, Lu Fang, Leye Wang, Ge Li, Bing Xie, and Fuqing Yang. 2011. APIExample: An effective web search based usage example recommendation system for Java APIs. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 592–595.

[31] Yi Wei, Nirupama Chandrasekaran, Sumit Gulwani, and Youssef Hamadi. 2015. Building bing developer assistant. (2015).

[32] Peter Weißgerber and Stephan Diehl. 2006. Identifying Refactorings from Source-Code Changes. In *IEEE/ACM Intl. Conf. on Automated Software Engineering*. IEEE Computer Society.

[33] Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. 2012. Example overflow: Using social media for code recommendation. In *Intl. Workshop on Recommendation Systems for Software Engineering*. IEEE Press, 38–42.

[34] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and recommending API usage patterns. In *ECOOP 2009–Object-Oriented Programming*. Springer, 318–343.