Software Developer Activity as a Source for Identifying Hidden Source Code Dependencies

Martin Konôpka, Mária Bieliková

Slovak University of Technology,
Faculty of Informatics and Information Technologies,
Ilkovičova 2, 842 16 Bratislava, Slovakia
{martin_konopka, maria.bielikova}@stuba.sk

Abstract. Connections between source code components are important to know in the whole software life. Traditionally, we use syntactic analysis to identify source code dependencies which may not be sufficient in cases of dynamically typed programming languages, loosely coupled components or when multiple programming languages are combined. We aim at using developer activity as a source for identifying implicit source code dependencies, to enrich or supplement explicitly stated dependencies in the source code. We propose a method for identification of implicit dependencies from activity logs in IDE, mainly of switching between source code files in addition to usually used logs of copy-pasting code fragments and commits. We experimentally evaluated our method using data of students' activity working on five projects. We compared implicit dependencies with explicit ones including manual evaluation of their significance. Our results show that implicit dependencies based on developer activity partially reflect explicit dependencies and so may supplement them in cases of their unavailability. In addition, implicit dependencies extend existing dependency graph with new significant connections applicable in software development and maintenance.

Keywords: software component, dependency, source code, developer activity, dependency graph, implicit dependency, implicit feedback

1 Introduction

Source code dependencies traditionally reflect explicit statements in the source code and are identified with syntactic analysis of source code contents. As a dependency we understand oriented connection between two source code components of selected granularity, namely instance or type reference, inheritance relationship or call reference.

Identified dependencies are used to form a dependency matrix or an oriented graph of interconnected software components to study organization and hierarchy of software components and their attributes [7]. Dependencies are also sourced for identifying problematic places, possibly code smells and complexity of the web of software components, which is important for maintenance activities on evolving software in particular.

Source code dependencies allow software developers to learn about how the existing source code works and how it is composed, e.g., how it will be affected by an introduced

change or how much effort will be required for refactoring. Both adding new functionality and changing existing functionality require developers to know about the dependencies before making a change in the source code.

Traditional approaches use a syntactic analysis for identifying source code dependencies. Other approach is to employ developer activity as a source for identifying source code dependencies, but mostly logs of copy-pasting code fragments and commits to identify hidden dependencies in source code are used. We propose a method for identification of source code dependencies that extends existing works with utilizing data of developer's navigation in source code space (logs of switching between source code components). Based on the source for identification, we distinguish identified source code dependencies as *implicit*, i.e., identified from developer activity as an implicit feedback related to the source code, in addition to the traditional *explicit* dependencies reflecting explicit statements in the source code. With our method we broaden the space of known source code dependencies, thus extending a dependency graph with new edges relevant for the development or other evaluations of the source code.

For the identification of the implicit source code dependencies we use developer activity recorded in an integrated development environment (IDE) and commit logs from a revision control system (RCS) [14]. Our work is inspired by the research project PerConIK – Personalized Conveying of Information and Knowledge (perconik.fiit.stuba.sk) [2] with its goal to bring new software metrics based on evaluating data of developer activity and context of software development. Infrastructure of this project [3] provides us with data collected in software house and university environment (student team projects), which we use for evaluation of our method.

2 Related Work

Software product and its source code result from software developer activity. This motivates current research to look for how software attributes (mostly maintainability) [4] are affected by activities performed during the development together with the context which developers had resided in. Developers are often disrupted at work, switch between multiple tasks or take over another developer's task. Because of that, various tools for navigation in the source code were proposed, notably dependency graph of software components [7] and task-related tools, e.g., for source code and developer recommendation [1], [6], [13].

We may infer programming sessions [6] from developer activity monitored during the software development and use them to describe tasks which developers had worked on with task contexts [1], [15], i.e., source code artifacts relevant to the currently solved task by developer. Developers visit places in the source code related to the task more often during the work on that particular task [4], [8]. From the recorded data we may then reconstruct what the developer was working on [6], [13] or what particular developer specializes in [11].

There are multiple sources and types of data that we can gather when monitoring developers during the processes of software development [14], for example: source code files and their contents from source code repositories [13]; development tasks

from issue tracking systems; developer activity in an IDE [1], [3], [7]; developer activity outside of an IDE, in operating system or even events in real life.

It is important to not affect monitored developers during their activities [8], being it a similar problem of gathering implicit feedback on the Web [2]. Several issues arise in the design and development of the infrastructure for a system for gathering mentioned data of developer activity together – scalability and efficient processing online among them. One of the already proposed solutions is developed within the project PerConIK – Personalized Conveying of Information and Knowledge which considers software repository as a web of software components and applies "webification" of software development [2], [3], i.e., employing methods and techniques from Web engineering to identify new information about software development and propose new software metrics.

Traditional source code metrics rely on a syntactic analysis of source code, omitting the information from development process. Basic example is the *lines of code* metric which evaluates the size of source code but not the time spent working on the measured source code. Similarly, traditional dependency graph of software components is created with identified references of source code components [7], helping developers with software development and maintenance. Authors in [17] also successfully applied network algorithms on identified dependencies to predict problems in software design.

Dependencies identified with syntactic analysis of source code are *explicit* since they reflect explicit statements in the source code [7]. We identify following main problems of the explicit dependencies and of their identification:

- Explicit dependencies do not capture cross-language connections in source code, e.g. in Web projects developed in combination of HTML and C# language.
- Explicit dependencies do not capture connections with configuration files, schema template files or runtime dependencies.
- Syntactic analysis of source code is not trivial or even possible for dynamically typed languages, e.g., JavaScript, Ruby.
- Explicit dependencies do not reflect sources of solutions in source code, developer's inspiration and places required to check when particular component is changed.

We see possibilities of employing developer activity as a source for identification of source code dependencies, inspired by the task context approaches [1], [6], [13]. Source code does not contain information about the developer's intents, inspirations and decisions for implemented solutions, what may suitably extend existing dependency graph. Moreover, because developer activity is not language-dependent, we may identify dependencies across different programming languages and also dependencies with configuration files or others which are currently not covered by explicit dependencies.

3 Implicit Source Code Dependencies

Software developer interacts with source code components during the development and maintenance, and so implicitly reveals task-related dependencies hidden in the source code. Selected example situations are:

- Developer studies existing code and navigates between dependent components, switches between them to understand implemented logic (navigation paths) [8].
- Developer implements functionality consisting of multiple components in parallel.
- Developer copy-pastes a code fragment from existing component – further changes of the original implementation may lead to inconsistency in source code [4].
- Developer configures source code components by creating or maintaining external configuration files.

In all these situations developer's navigation and activity performed in the source code implicitly reveals dependent components. When we do not take into account contents of source code files, developer activity also reveals dependencies on configuration files or on components implemented in different programming language. The idea behind the implicit dependencies is similar to the identification of task contexts, i.e., the developer works with software components that are related with each other for the task completion, and is based on empirical observations of developer activity from two sources – activities recorded in an IDE, e.g., custom extension for Microsoft Visual Studio or Eclipse [3], and commit history in a RCS, e.g., Microsoft Team Foundation Server or Git [12]. We chose to use these low-level logs of activities with source code components from all available types of logs [14] provided by the project PerConIK [2]:

- navigation in the source code in an IDE – open, close and switch-to a source code file – time-related activities resulting in a change of currently opened file;
- copy-paste code fragment between two source code files; and
- commit (or check-in) of a collection of source code files to a RCS.

Although we do not force choice of the granularity for source code components (e.g., line of code, method, class or library), but for our experiments and implementation we consider source code files as components.

Our method (see Fig. 1) consists of steps for converting raw logs of developer activity in an IDE and from a RCS to the format used by our method, continued by the identification of implicit dependencies, their weighting and validation and finally construction of a dependency graph.

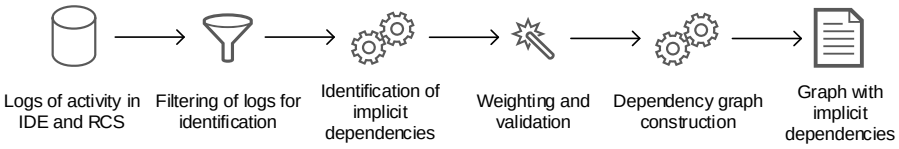

Fig. 1. Overview of method for identification of implicit source code dependencies.

3.1 Identification of Implicit Dependencies

We define source code dependencies as oriented connections between pairs of software components. Dependencies are weighted according to their significance and, considering software evolution, are valid for the particular time. Let *S* be the set of source code components of the selected granularity, then the space of dependencies in the source

code is $D=S \times S \times T \times R$ in the time T with the weights R. Note that the explicit dependencies are valid in time while explicit statements are present in the source code.

Based on the types of activity logs used for identification of the implicit dependencies we define three specialized types of implicit dependencies $D_{imp} \subseteq D$:

- time-related implicit dependencies $D_{imp,time}$,
- content-related implicit dependencies $D_{imp,content}$,
- commit-related implicit dependencies $D_{imp,commit}$.

The most common activity performed by developer during the development is the navigation in the source code space. Ttime-related activities with source code components in an IDE are described with the tuples *(source, target, operation, timestamp)* containing the *source* and the *target* component of the operation, *type* of performed operation (e.g., open, close or switch-to file) and *timestamp* when the activity occurred. We reconstruct developer activity from these logs to create time-related implicit dependencies $d_{imp,time}$ (1) between the software components s_1 and s_2, which occurred at the time *t*, when developer spent time span in the target component (the *time window* property) before making next operation. The weight *w* is determined by the weighting function using the *time window* property.

$$d_{imp,time} = (s_1, s_2, t, w, time\ window) \tag{1}$$

Content-related activities of copying and pasting code are described with the tuples *(target, code operation, content, timestamp)*, containing the *target* component where the *code operation* was performed (copy, cut or paste) with the code *content*, and when the operation happened. Final copy-paste operation is logged with at least two actions, i.e., copying the code from the source code component *X* and pasting it into the source code component *Y*. Because of that we reconstruct the clipboard stack to identify content-related implicit dependencies $d_{imp,content}$ (2) where the *content* property contains the copy-pasted code fragment and may be used for determining the weight *w*.

$$d_{imp,content} = (s_1, s_2, t, w, content) \tag{2}$$

The last type of implicit dependencies are identified from commit operations. Developers tend to submit changes in a collection of source code components as a solution for the particular task, thus the changed components are implicitly connected with each other. For each pair of the changed components (s_1, s_2) in the same commit we create dependency $d_{imp,commit}$ (3) with *total count* of all committed components and weight *w*.

$$d_{imp,commit} = (s_1, s_2, t, w, total\ count) \tag{3}$$

3.2 Weighting of Implicit Dependencies

Each specialized type of implicit dependency extends general D_{imp} with extra property, being it *time window*, *content* or *count*. We use these properties to determine the significance of dependencies with the weighting function *weight* differently for each specialized type (4), ranging from insignificant to fully significant dependency.

$$weight : D_{imp} \rightarrow \langle 0,1 \rangle \quad (4)$$

Time-related implicit dependencies are weighted according to the time spent in the visited component, i.e., significance of visiting (opening, switching-to) that component for the developer. The weighting function may be specified for the particular implementation. In our case we chose the weighting function to be as shown in Fig. 2. To eliminate mistakes in developer's navigation in source code space, the dependency becomes fully significant after the selected threshold *a*. But after the threshold *b* the dependency is becoming irrelevant (the threshold *c*). After experiments we chose the thresholds to be *a = 10 seconds*, *b = 10 minutes* and *c = 15 minutes* for our method.

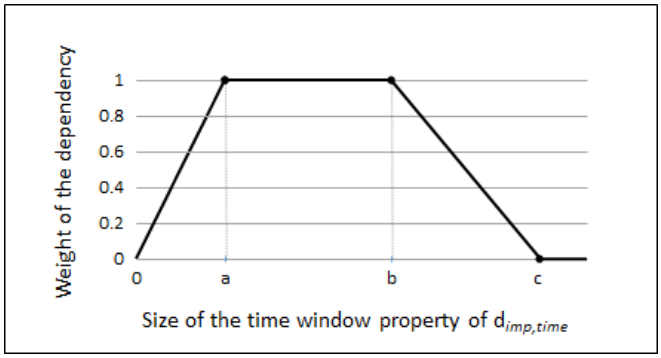

Fig. 2. The weighting function for time-related implicit dependencies $d_{imp,time}$ with threshold parameters *a*, *b* and *c*.

Content-related implicit dependencies are identified from the copy-paste operations, thus their significance may correspond to the amount of the code copied or its contents. However, for simplicity we chose to weight every $d_{imp,content}$ with constant of *1*.

Commit-related implicit dependencies are weighted to the total count of committed components, as in (5), to promote smaller, fine-grained, commits solving single tasks. The lower the count of changed components in a commit, the more dependent they are together and contrariwise.

$$weight(d_{imp,commit}) = \frac{1}{count} \quad (5)$$

3.3 Validation of Implicit Dependencies

Even though changes in the source code invoke changes of its explicit dependencies, we are not able to similarly validate implicit dependencies over time. To model their relevance we validate them to the selected time using a forgetting function [9]. Developer interacts with the source code components mostly based on the task they currently solve, thus when the contents change over time or the task is finished, the developer's interactions lose their significance. Because of that the definition for validation functions of implicit dependencies D_{imp} (6) is similar to of explicit dependencies D_{exp} (7).

$$validity_{imp} : D_{imp} \times T \rightarrow \langle 0, 1 \rangle \quad (6)$$

$$validity_{exp} : D_{exp} \times T \rightarrow \{0, 1\} \quad (7)$$

Explicit dependencies are valid or not according to existence of explicit statements to the selected time, hence the validity selected to 0 or 1. Implicit dependencies are valid according to the forgetting function to the selected time t (8), which we chose to use from [16] with parameters set to $a = 1$ and $b = 0.08$:

$$y = ae^{(-b\sqrt{t})} \quad (8)$$

3.4 Dependency Graph

After the identification of implicit dependencies, we extend existing dependency graph *G(V,E)* of *V* for the vertices of software components (source code files), and *E* for edges of aggregated dependencies. We differentiate between explicit and implicit edges in the graph because of differences in their meaning, i.e., $E = E_{exp} \cup E_{imp}$:

- Explicit dependencies represent statements in the source code, e.g., references, call hierarchy, inheritance.
- Implicit dependencies represent how developers interacted with source code during the development, e.g., their inspirations.

Both explicit and implicit edges of dependency graph are weighted with sum of weights of aggregated explicit or implicit dependencies of the corresponding type (9), possibly validated to the selected time. Resulting weight of implicit edge represents significance of aggregated implicit dependencies over time with single value.

$$weight(e_{imp,s_1,s_2}) = \sum_{d_{imp} \in D_{imp,s_1,s_2}} validity(d_{imp})w \quad (9)$$

4 Evaluation

We propose implicit source code dependencies to be mainly applied during the processes of development and maintenance. To evaluate contribution of implicit dependencies we performed two experiments to show that:

- implicit dependencies reflect explicit dependencies of components which developer worked on during the task,
- implicit dependencies enrich dependency graph with new significant connections usable during the maintenance.

The PerConIK project is developed in cooperation with a medium sized software company. Before deploying our method in real work environment, we opted for evaluating it using data of five on-going student software projects provided also by the PerConIK project. These projects were developed by students of master courses Software Engineering or Information Systems. All projects were developed in C#/.NET and other Microsoft technologies in time span of one academic year:

- *Project A* – development of a web/desktop application by 3 developers,
- *Project B* – development of class libraries by 1 developer,
- *Project C* – development of class libraries by 1 developer,
- *Project D* – development of a web/desktop application by the team of 7 developers,
- *Project E* – development of a web application by the team of 7 developers.

Table 1 shows total numbers of activity logs from these projects, total number of explicit edges in their dependency graphs and also results of the selected source code metrics – *lines of code*, *maintainability index* and *cyclomatic complexity*. In our evaluation we use source code files for components used in the method. Because all the evaluated projects were developed using Microsoft technologies, we employed the Code Map functionality of Microsoft Visual Studio to identify explicit dependencies using the built-in reference recognition and the Code Metrics to evaluate other metrics.

4.1 Reflection of Explicit Dependencies

We expect that implicit dependencies reflect explicit ones based on the existence of the developer's task context. We compared existing explicit dependencies of evaluated projects with identified implicit dependencies by comparing the E_{exp} and E_{imp} sets of dependency graphs. Table 2 shows total numbers of identified implicit edges in dependency graphs constrained by threshold for edge weights ranging from 0 to 3, when considering only edges between the files that appear in the explicit dependency graphs.

Table 1. Results of the source code metrics for the software projects used for our evaluation, total numbers of explicit edges in their dependency graphs and numbers of activity logs recorded in 1 year of their development (LOC – lines of code, MI – maintainability index, CC – cyclomatic complexity).

Project	LOC	MI	CC	$\lvert E_{exp} \rvert$	No. of activity logs
A	2,402	82	1,285	232	15,215
B	4,384	74	2,164	274	8,584
C	4,993	81	3,213	528	24,213
D	5,342	81	1,839	270	55,877
E	3,721	81	1,779	189	48,717

Table 2. Total numbers of edges for identified implicit dependencies in dependency graphs for the evaluated software projects with the same vertices set.

Project	$\lvert E_{imp} \rvert$ for thresholds			
	0	**1**	**2**	**3**
A	640	423	200	124
B	507	339	141	88
C	792	524	224	124
D	797	556	285	201
E	755	464	246	164

For evaluating the overlap of implicit and explicit edges we used weight thresholds of 1 and 2 to filter out less significant implicit edges. Higher values may have been also used but not for the dataset of size of ours. Table 3 shows how much of the identified implicit edges are explicit as well and how many explicit edges were identified with implicit edges (last column). We found out that up to 54% of all the identified implicit dependencies are common with explicit dependencies. We also analyzed whether the rest of the implicit dependencies are significant or not (see section 4.2).

Table 3. Overlaps of implicit and explicit edges in dependency graphs with selected thresholds to filter out less significant implicit edges.

Project	E_{imp} threshold	$\lvert E_{exp} \cap E_{imp} \rvert$	$\frac{\lvert E_{exp} \cap E_{imp} \rvert}{\lvert E_{imp} \rvert}$	$\frac{\lvert E_{exp} \cap E_{imp} \rvert}{\lvert E_{exp} \rvert}$
A	1	173	40.90%	74.57%
	2	108	54.00%	46.55%
B	1	140	41.30%	51.09%
	2	70	49.65%	25.55%
C	1	210	40.08%	39.77%
	2	120	53.57%	22,73%
D	1	191	34.35%	70.74%
	2	123	43.16%	45.56%
E	1	149	32.11%	78.84%
	2	108	43.90%	57.14%

Secondly, identified implicit dependencies overlapped up to 78.84% of explicit dependencies between files included in the space of implicit dependencies. That means that we are able to partially compensate inability of identification of explicit dependencies in cases where source code analysis is not possible but monitoring of developer activity is. Such example is usage of multiple programming language in the same software project, e.g., when simply combining HTML, JavaScript and CSS.

4.2 Significance of Implicit Dependencies

For the second experiment we expected that implicit dependencies provide new and significant information about connected software components, e.g., when particular component in a source code file relies on settings in a configuration file or when components are loosely coupled. Our task was to discuss identified implicit dependencies with the developers and decide whether they reflect connections in source code usable in the maintenance or not. As a significant connection of source code files, i.e., significant implicit dependency, we understood: *If the components in the source code file A are changed, the contents of the file B should be checked or changed as well.*

We were able to perform this experiment on the first four projects only and we chose to validate implicit dependencies with the weight threshold of 2. We chose this threshold to evaluate as most of the implicit edges as possible while still keeping the number

of edges relatively low and the experiment bearable for developers. Developers manually checked each dependency and decided its significance, hence evaluating all the identified edges would have been too difficult. In this experiment the counts of implicit edges were higher than in the first experiment because we also evaluated dependencies between files which were not included in the explicit dependency graphs, e.g., webpages, configuration files, etc. To simplify the evaluation process, we generated dependency graphs in DGML format (for Microsoft Visual Studio) with implicit edges only, excluding the explicit ones. Developers were able to switch between the graph and the file, thus ensure in their decision of keeping or removing the dependency from the graph. In the end we compared the results with the original files. We achieved precision of more than 75% for the evaluated projects (Table 4).

Table 4. Evaluation of significance of implicit dependencies with the weight threshold of 2.

Project	No. of implicit dependencies		Precision
	Original	Significant	
A	180	138	76.67%
B	112	103	91.96%
C	257	203	78.99%
D	634	576	90.85%

During the experiment we led discussion with the participated developers and received positive feedback for ability to identify dependencies between loosely coupled components across layers of the Model-View-Controller pattern. Even more, developers reminded reasons why the files were dependent during their work with them.

We highly appreciate the ability to identify dependencies just from logs of switching between source code files in the IDE. This is important when looking for dependencies on configuration files, schema template files or dynamically resolved dependencies. As an example, these situations were correctly identified with our method from developer activity in the IDE (for Web projects in ASP.NET MVC):

- Dependencies between C# classes and the XML configuration files, e.g., key-value settings, database connection strings, web service definitions.
- The View layer components (HTML webpages) displaying contents of the Model layer components (C#), e.g., webpage displaying data of a data class in table view.
- Dependencies between the View layer components (HTML webpages) on the Controller layer components (C#), e.g., when linking to a controller action.
- Dependencies between JavaScript source code files and C# files.
- Correct pairings of the View layer (HTML webpages) with its code-behind (C#).
- Transitive dependencies on class inheritance hierarchies through class interfaces.

5 Conclusion

Knowledge about dependencies of software components is utilized mostly during the development and maintenance processes, helping software developers with navigation

and study of the existing source code. However, identification of explicit dependencies does not provide information about all connections in the source code. Moreover, in case of dynamically typed languages, it is sometimes even impossible to identify dependencies at all. Because of that we proposed the identification of implicit dependencies from developer activity to enrich existing dependency graph with new significant edges, or to supplement explicit dependencies in case of their unavailability.

For the evaluation of our method we used data gathered in the course of student (team) software projects. While we see natural difference between behavior of students and professional developers (work habits and schedule, experience), the evaluated projects were of relatively large size considering school projects and served as a basis for next step in our research, which aims at evaluating our method in real work environment. In our first experiment we showed overlap of both explicit and implicit edges in dependency graphs, thus possibilities of supplementing explicit dependencies with implicit ones. In our second experiment we attempted to manually evaluate significance of identified implicit dependencies. We achieved positive results, with correctly identifying hidden dependencies in the source code, and also cross-language dependencies.

Achieved results allowed us to deploy our monitoring infrastructure to a medium size software company. The infrastructure is aimed at recording implicit feedback of software developers and annotating source code with information tags created manually by the developers (during code reviews) or automatically based on source code analysis and developer activity analysis [3]. In June 2014 we have started to record activity data from two teams of total 25 developers working on web information systems development in this software company. Just before the deployment of our developed infrastructure within the PerConIK project for recording implicit feedback of software developers we tested the infrastructure extensively.

First impression of developers on the dependencies enriched by implicit dependencies was very positive including examples of such dependencies identified even by hand in existing software repositories. While the straightforward application of implicit dependencies is to visualize them in the form of a graph, we discussed our method with professional software developers and received valuable feedback to simply provide prioritized list of software components to be checked for the selected component upon developer's request. We plan to continue in experimental evaluation with dataset based on professional developers' work.

Acknowledgment. This work was partially supported by grants No. VG 1/0752/14 and it is the partial result of the Research and Development Operational Programme for the project Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF.

References

1. Antunes, B., Cordeiro, J., Gomez, P.: An Approach to Context-based Recommendation in Software Development. In: Proc. of the 6th ACM Conf. on Recommendation Systems, pp. 171-178. ACM (2012)

2. Bieliková, M., Návrat, P., Chudá, D., Polášek, I., Barla, M., Tvarožek, J., Tvarožek, M.: Webification of Software Development: General Outline and the Case of Enterprise Application Development. In: AWERProcedia Information Technology and Computer Science, Vol. 3: 3rd World Conf. on Information Technology, pp. 1157-1162 (2013)
3. Bieliková, M., Polášek, I., Barla, M., Kuric, E., Rástočný, K., Tvarožek, J., Lacko, P.: Platform Independent Software Development Monitoring: Design of an Architecture. In: Proc. of 40th Int. Conf. on Current Threads in Theory and Practice of Computer Society, LNCS 8327, pp. 126-137. Springer-Verlag (2014)
4. Bird, C., Nagappan, N., Gall, H., et al.: Putting It All Together: Using Socio-technical Networks to Predict Failures. In: 20th Int. Symposium on Software Reliability Engineering, pp. 109-119. IEEE CS Press (2009)
5. Boehm, B.W., Brown, J.R., Lipow, M.: Quantitative Evaluation of Software Quality. In: Proc. of the 2nd Int. Conf. on Program Comprehension, pp. 592-605. IEEE CS Press (1976)
6. Coman, I.D., Sillitti, A.: Automated Identification of Tasks in Development Sessions. In: Proc. of 16th IEEE Int. Conf. on Program Comprehension, pp. 212-217. IEEE CS Press (2008)
7. Counsell, S., Hassoun, Y., Loizou, G., et al.: Common Refactorings, a Dependency Graph and Some Code Smells: An Empirical Study of Java OSS. In: Proc. of the ACM/IEEE Int. Symp. on Empirical Software Engineering, pp. 288-296. ACM (2006)
8. DeLine, R., Czerwinski, M., Robertson, G.: Easing Program Comprehension by Sharing Navigation Data. In: Proc. of the 2005 IEEE Symp. on Visual Languages and Human-Centric Computing, pp. 241-248. IEEE CS Press (2005)
9. Ebbinghaus, H.: Memory: A Contribution to Experimental Psychology, transl.: Ruger, H.A., Bussenius, C.E. New York: Teachers College (1885/1913)
10. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach. 2nd Edition, PWS Pub. Co., Boston, MA, USA (1998)
11. Fritz, T., Murphy, G.C., Hill, E.: Does a Programmer's Activity Indicate Knowledge of Code?. In: Proc. of 6th Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. on The Foundations of Software Eng., pp. 341-350. ACM (2007)
12. Kalliamvakou, E., Gousios, G., Spinellis, D., et al.: Measuring Developer Contribution from Software Repository Data. In: Proc. of the 4th Mediterranean Conf. on Information Systems, pp. 600-611. Greece (2008)
13. Kersten, M., Murphy, G.C.: Using Task Context to Improve Programmer Productivity. In: Proc. of 14th ACM SIGSOFT Int. Symp. on Foundations of Software Eng., pp. 1-11. ACM (2006)
14. Polášek, I., Ruttkay-Nedecký, I., Ruttkay-Nedecký, P., Tóth, T., Černík, A., Dušek, P.: Information and Knowledge within Software Projects and Their Graphical Representation for Collaborative Programming. In: Acta Polytechnica Hungarica, vol. 10, no. 2, pp. 173-192. ISSN: 1785-8860 (2013)
15. Robillard, M.P., Murphy, G.C.: Automatically Inferring Concern Code from Program Investigation Activities. In: Proc. of 18th IEEE Int. Conf. on Automated Software Engineering, pp. 225-234. IEEE CS Press (2003)
16. White, K.G.: Forgetting Functions. In: Animal Learning & Behavior, Vol. 29, No. 3, pp. 193-207. Springer-Verlag (2001)
17. Zimmermann, T., Nagappan, N.: Predicting Defects Using Network Analysis on Dependency Graphs. In: Proc. of 30th Int. Conf. on Software Engineering, pp. 531-540, ACM (2008)