Empirica Metadata Maintenance in Source Code Development Process

Karol Rastocny
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Bratislava, Slovakia
Email: karol.rastocny@stuba.sk

Maria Bielikova
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Bratislava, Slovakia
Email: maria.bielikova@stuba.sk

***Abstract*—A management of software development process is a crucial part of the software engineering, from which the success of software projects is dependent. This management mostly relays upon quality and freshness of software metrics, especially empirical software metrics. But empirical software metrics are sensitive to source code modications and also to developers activities. In this paper we proposed approach for a maintenance of empirical metadata stored in information tags. The approach covers main maintenance actions - creating missing, repairing invalidated and removing unrepairable informationtags.Themaintenanceisprovidedviataggingrules executed after matching predened templates in a stream of developersactions.Wealsointroducesasetofinformationtags for supporting software development process and we describe their utilization in teaching a subject Team project.**

Keywords-information tag; metadata management; empirical metadata; developers activities; software metrics; code review

I. Introduction

Analysing and understanding history of a software project development is important for project managers and developers. They spend a lot of time by an analysis why some events have occurred and by proposing processes with goals to prevent negative events and also to repeat positive ones. Managers and developers often nd this activity more important than forecasting next project evolution [3] . During this activity they utilize software metrics based on source code (e.g., LLOC) and behaviour of developers (empirical software metrics - software metrics derived from observations of developers activities). Nowadays, especially codebased software metrics are often used. It is because of a number of approaches and availability of source code, which allows calculation of metrics in the time, when they are required. But even if we calculate code-based metrics across change-sets (commits) we have still only static information about source code with results of development decisions and we are not able to nd an answer to the Why questions - Why is the source code erroneous? Why was the projects time-plan exceeded? ...

These questions can be answered after a consideration in a wider context which can be brought by empirical software metrics. For example empirical software metrics give to us information as:

- Cooperation of developers during problem solving;
- Used information or knowledge sources (e.g. webpages that has been read by a developer);
- Disturbing impacts (e.g. received emails, freezing up of a computer due to background processes).

Even though empirical software metrics are not widely used. Most of software project managers used only basic information from systems for development aid (e.g. plan fullment from issue tracking systems, productivity exported from source control systems) or in occasional cases collaboration of developers extracted from discussions (e.g. discussions in issue tracking systems [13]). This is caused by three problems [7]:

- *Collecting of empirical data are time and resource expensive* - many data are collected manually by developers (e.g. an information about a problem consulting with another developer);
- *Quality of collected data* - manually collected data are often erroneous and developers forgets to store some events;
- *Usability of empirical data* - there are lack of empirical software metrics and tools for their interpreting and analysis.

Partial solutions of these problems had been proposed by several systems that tried to collect empirical data via tools installed on developers machines [7], [19] or automatic watching collaboration of developers and moving around a software house by Bluetooth transmitters and watching tools installed on developers cellphones [4]. Advantages of these approaches is non-disturbing collecting data. But they do not directly interconnect collected empirical data with source code and also they proposed only limited tools for empirical data analysis (at most they contain a framework, in which third-party tools can be executed).

The solution of the problem of lack of metrics and utilities can lay in utilizing approaches and methods from web engineering that analyse and use empirical data (e.g. keyword-based search query sequences and web-page visits for results recommendation [10]). If we look on the information space of a software house as on a spider-web of software artefacts, in which relations between artefacts are their dependencies at different levels of an abstraction, we can modify and reuse web engineering methods and approaches in a domain of software houses (e.g. for an enhancement of search in source code [11] or for identication of developers' tasks in their com-

mits [9]). We utilize this principle in the project PerConIK[1] (Personalized Conveying of Information and Knowledge) which is focused on a support of enterprise applications development in a software house via empirical software metrics and their application in software development processes [2] . In the project PerConIK we collect empirical data via software tools and extensions for integrated development environments (Microsoft Visual Studio 2012 and Eclipse) and web browsers (Mozilla Firefox and Google Chrome) installed in developers working environments, that collect developers activities as open/add/edit source code le, copy/paste and visited web-pages and biometric information (e.g. intensity of mouse clicks).

We store developer-oriented empirical data and software metrics in form of information tags [15] that are directly related to software artefacts. But empirical data are error prone,whiletheirvaliditycanbeaffectedbyvarioussources. The rst of all is instability of source code during development process. But validity of information tags can be affected by other empirical data as activities of developers or time, too. In this paper we describe basics of information tags and their dependencies (Section II) and basics of proposed approaches for maintenance of empirical software metrics anchored to software artefacts via information tags with rst evaluations (Section III). Proposed methods in these sections address problems of collecting and quality of empirical data. In Section IV we present real utilization of information tags in software development process with a set of used information tags via which we present example of utilization of empirical data. The last section concludes the papers contributions and discuss possibilities of a future work.

II. CONCEPT OF INFORMATION TAGS

The concept of information tags is based on simplicity of conventional tags used in folksonomies as forms of a lightweight ontology [5]. Folksonomies utilize tags (keywords) as basic entities of an objects description. In our manner, an information tag is not only a simple keyword assigned to an object but it is a structured information which has been assigned to the object for a specic, dened reason (we can say, that information tags map structured data to theirresources,taggedobjects,withsemanticrelations).This categorize information tags to scope of descriptive metadata (based on NISO classication [14]) that are dened by triples of a type, an anchoring and a body [2].

Information tags can have different nature which is dependent on their sources (a creator and source data) and consumer (software systems or users). Thus we classify information tags as follows:

- *Machine information tags* - information tags that are created by software systems. Machine information tags can be more precisely classied to:
  - *Content-based information tags* - information tags directly inferred from a tagged content, i.e. codebase software metrics;
  - *User activity information tags* - information tags inferred from users activity over tagged resources, i.e. empirical software metrics. These information tags are important because they can be used for a detection of hardly obtainable information as popular or frequently visited source code;
  - *Aggregating information tags* - information tags inferred from another information tags, e.g. an average ranking of a source code. These information tags simplify information stored in multiple information tags so they can be directly used as analytical results for developers and managers. Aggregating information tags can additionally contain partial results that are used by multiple software systems or tools that directly enhance performance of computation (e.g. both tools for searching in source code and tools for ranking of developers use average source code ranking. They do not have to process all ranking information tags and calculate average rankings, but they can directly read an aggregating information tag with pre-calculated average ranking);
- *User information tags* - information tags that are created by users. Even though user information tags are not created so frequently as machine information tags created by software systems, these user information tags contain users knowledge and opinions about content. Examples of users information tags are rankings, formal tags or structured annotations created by fullling of predened forms.

Information tags are closely related to resources at two levels of an abstraction. The lowest level is anchoring, which exactly identies tagged source code artefact. So anchoring is directly dependent from structure and syntax of source code and no empirical events or modications of source code semantics have any inuence to anchoring. We have solved problem of anchoring dependency by proposition of the robust location descriptor for source code which provide acceptable accuracy and can be interpreted in real time [16]. This descriptor gives us a possibility to anchor information tags to words (commands) of source code.

Higher level of relation between information tags and source code is provided via bodies of information tags. Bodies contain main knowledge or information about tagged source code, so they can be affected not only by structure of tagged source code, but their validity is dependent on semantic of source code and empirical events (e.g. developers activities) too. It depends on information tags types and sources, which of structural, semantic or empirical properties of source code affects validity of information tags (see Table I). Content-based information tags are inferred directly from structure and/or semantic of source code and they have no empirical dependencies. On the other side, information tags based on user activity are closely dependent on empirical data. In case of

[1] http://perconik.fiit.stuba.sk

TABLE I
DEPENDENCIES OF INFORMATION TAGS CLASSES ON STRUCTURAL AND SEMANTIC PROPERTIES OF SOURCE CODE AND EMPIRICAL DATA (DEVELOPERS ACTIVITY) RELATED TO SOURCE CODE.

Information tags	Structure p.	Semantic p.	Empirical data
Content-based	X	X	
User activity		X	X
Aggregating	X	X	X
User	X	X	(X)

aggregating information tags, these dependencies are inherited from information tags that are aggregated.

User information tags have special position. User information tags are mainly dependent on the structure and the semantic of source code, because developers tag source code with their opinion on source code and they do not know any empirical data about source code (they do not know who and how interacts with the code). But their tags are indirectly dependent on empirical data, too. This is caused by human factor - developers tag source code with their individual knowledge in some context (previous activity, tiredness, etc.)

III. EMPIRICAL METADATA MAINTENANCE

All information tags are closely related to source code (all information tag classes needs at least minimal knowledge about source code semantic - see Table I) and it seems to be necessary to analyse source code for maintenance information tags. On the other side, all information tags for up to content-based information tags are dependent on empirical data and also source code is created in some context (denable by empirical data) which inuences source code properties. Therefore it should be possible to substitute structural and semantic source code properties by a set of empiricaldataandinformationtagmaintenancecanbebased almost only on empirical data collected during development process.

A. *Creating, Repairing and Removing Information Tags*

Main goal of information tags maintenance is to keep information tag space valid and consistent. Simply, similar source code artefacts should be described by similar sets of information tags. To full these requirements, the maintenance has to provide following maintenance actions [2]:

- Create missing information tags;
- Repair invalidated information tags;
- Remove unrepairable information tags.

Based on aforementioned analysis of information tag dependencies we can identify three data sources that directly inuence validity and consistency of information tag space - source code, developer-oriented empirical data and other information tags. Following these sources we can also dene events that execute process of information tags maintenance:

- *Commit* - when developers nished editing source code, they publish change-sets over source codes by commits so information tags with code-based software metrics can be recalculated;
- *Receiving empirical data* - every time when empirical data are received, processes that calculate empirical software metrics should be executed;
- *Modication in information tags space* - aggregating information tags aggregate data from other information tags, so each modication in information tags space (adding new, editing or removing old information tag) can lead to recalculating aggregated information which use modied information tags data;
- *Time* - some information tags are time dependent and contrary they are created when aforementioned events does not arrive, e.g. information tags that identify source code which has not been used for long time.

Therefore, executing maintenance actions can be event-oriented whereby it should be enough to watch stream of aforementioned events and to wait for a match of a template of an event sequence which indicates, that an information tag should be created. The main contribution of this idea is that we do not have to store and reprocess all data collected from developers working environments, but it is enough to store onlydatafromtimewindowwhichisnecessaryformatching event sequence template. As a result, software systems that create information tags do not have to deal with problem of big data analysis via querying a repository of all collected empirical data.

We implemented this idea in a proposition of a tagger [17] based on processing linked stream data [18]. An employing linked stream data gives us possibility to utilize advantages of linked data inference with a combination of simple specication of event sequence templates in a SPARQL-like language. The proposed tagger contains four main parts (see Figure 1):

- *Repository of tagging rules* - the tagger maintains information tags based on tagging rules that have specied stream queries and tagging actions (each rule has one query and one tagging action). Stream queries are used for matching sequences of events in the stream and selecting queried data. Following of a positive evaluation of queries, tagging actions uses queried data and create new information tags;
- *Linked stream data generator* - data collected from developers working environments (source code changesets and empirical data) and data about modications in information tags space are not directly received in form of linked data and they have to be transformed from structured objects to RDF graphs (used by Linked Data initiative [6]) and streamed out to Linked stream data processor. This task is provided by Linked stream data generator, which periodically connects to a source code repository, an empirical data collector and the repository of information tags and reads new unprocessed data. After that it transforms these data to RDF graphs and streams them out;
- *Linked stream data processor* - received linked stream data are processed by a processor which has registered tagging actions queries. A processing linked stream data deals with many challenges that are mainly related to

dynamicity of data stream (e.g. establishing a size of a time window, in which are queries processed) and optimization of queries [12]. Because the main contribution of our approach is not in linked stream data processing but in an employing of developeroriented empirical data in software development, we do not propose our own linked stream processor but we process linked stream data via C-SPARQL engine [**?**], [1]

- *Tagging actions executor* - queries results (selected data) are collected by a tagging action executor, which loads tagging actions of tagging rules, whose queries has been evaluated successfully, attaches queries results to tagging actions and executes them. As a result tagging actions create and add new information tags to information tags repository.

Possibilities of the tagger are limited only by expressiveness of tagging rules and his performance (amount of RDF triples that can be processed by the tagger). We proposed tagging rules as C-SPARQL query/action pairs. CSPARQL query can dene time dependencies and templates ofwatchedeventssoweshouldbeabletocatchallnecessary event sequences. The action part of the rule species type of maintenance activity (add, repair and remove) and identies maintained information tags by one of:

- *Information tag template* - a template of an information tag that is maintained. Information tags attributes can be expressed by literal values, parameters of CSPARQL query or simple expressions;
- *Executable code* - code which is executed for information tag maintenance. This code has possibility to provide more complex computations as simple expressions in information tag templates. Results of these actions are lists of maintained information tags.

To evaluate the taggers performance we deployed his prototype in the project PerConIK. We do not have nished complete evaluation yet, but in current setup the tagger daily processes around 11,000 events from 10 developers transformed to more than 100,000 RDF triples. To process these events the generator needs around 300MB RAM. These results are promising but we do not have evaluated time necessary to process standard C-SPARQL queries dened in tagging rules.

We also performed preliminary performance evaluation. For this evaluation we have registered C-SPARQL query which selects objects that consist of ve RDF triples and the query has been congured for 2s window. During the evaluation we repeatedly posted 500,000 RDF triples (100,000 objects) to Linked stream data processor and we measured response time of the processor (it means, that we measuredmillisecondsfromsendinganobjectsRDFtriplets to receiving information tags generated from the object). Because of we have 2s window, we should ideally obtain 1s response time in an average (2s for the rst object of the window and 0s for the last object in the window). In out testing environment (with 2x2.8GHz CPU), we reached this value (with 100ms tolerance), when we generated up to 300 triples per second

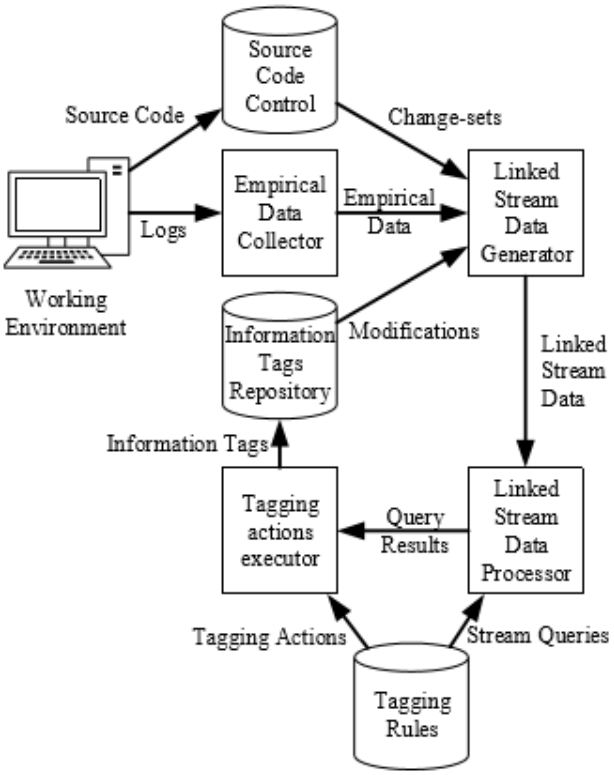

Fig. 1. The taggers data ow. Linked stream data generator collects new change-sets, empirical data and modications in information tags space and it transforms them to linked stream data that are consumed by Linked stream data processor, which processes tagging rules queries. Results of queries are processed by tagging actions in Tagging actions executor, that create new information tags.

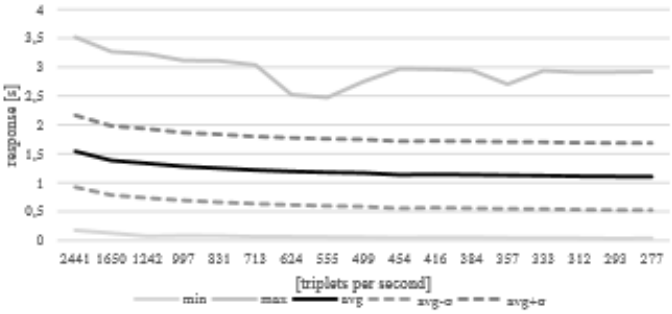

Fig. 2. Response times of the taggers testing environment in seconds per number of generated triplets. Ideal average value is 1 second.

(see Figure 2). This result proxi mately correspondents up to 800 developers. It seems to be unsatisfactory result, but by the analysis of partial results we nd out, that our bottleneck was caused by overhead of testing sandbox (the logger and the stream generator), that consumed more than 75% of resources. In planned release set-up, the stream generator and the processor do not share their resources. In addition each tagging rule is independent, so we are able to parallelize rules over multiple stream processors which can be deployed on cluster.

IV. IMPLEMENTATION OF INFORMATION TAGS IN SOFTWARE DEVELOPMENT PROCESS

We proposed the set of information tags for support of a software development process that covers all information tags classes:

- *Content-based information tags*
  - *Smell* - we collects smells from SonarQube's[2] analysis and attach them to source code les. This way we can easily reuse existing methods and tools of source code analysis and we do not lost currently used information.
- *User activity information tags*

  Implicit dependency - information tags that contains references to source code les on that source code les are implicitly dependent. We calculate these implicit dependencies from developers interaction with les. E.g. if developers often switch view from a le F1 to a le F2, than we can say that the le F1 is implicitly dependent from the le F2. These dependencies are especially important in dynamic programming languages like JavaScript. The original method of building implicit dependencies is described in [8].

  Author - tagged parts of source code with authors, developers that authored or edited the code (with precision to lines).
- *Aggregating information tags*
  - *Facet* - information tags that aggregate selected information from other information tags with goal to support faceted search of source code. E.g. information tag aggregates all authors from lines of a source code le to one attribute AuthorFacet as a set of authors, what allows fast search operations with authors of the le.
- *User information tags* - we proposed a set of user information tags from analysis of developers comments in source code les with goal to support code review process and motivate developers to create these information tags.
  - *ToDo* - a part of a source code should be implemented;
  - *Fix me* - tagged source code contains a bug which has to be xed;
  - *Nice to have* - tagged source code will be updated/reimplemented;
  - *To be done* - a work on the source code is in the progress;
  - *Sample* - tagged source code is the ideal solution of the problem and should be used as the sample code for similar problems;
  - *Review request* - the source code should be reviewed;
  - *Refactor* - the source code has to be refactored;
  - *Code review* - tagged source code contains a coding violation.

To competition employing information tags in software development we developed system CodeReview[3]. The system is focused to source code review, while it supports standard features of code review systems like browsing source code from git or requesting code review. In addition it supports direct tagging source code with aforementioned user information tags with precision to words (see Figure 3), interpreting machine information tags for users and faceted searching source code.

We employ the CodeReview system in teaching process of the subject Team project in which team of six or seven students develop software systems for one school year. We do not have nal results yet, but we observed increased quality of source code and faster acquiring of team management skills than in teams from previous years.

V. CONCLUSIONS AND FUTURE WORK

In the paper we presented concept, denition and classication of information tags and our approach for maintenance of empirical software metrics stored in information tags. The proposed approach has potential to solve the problem of collecting and processing empirical software metrics at real time,whatisnecessaryforfull-featuredemployingempirical software metrics in software development process. We have

not performed nal evaluation of the approach yet. But we have successfully implemented and deployed the rst prototype of the infrastructure already and we started with collecting empirical data and their processing to information tags. We also successfully implement information tags in teaching process. These facts suppose our assumption, that proposed approach is a solution for employing empirical software metrics in software houses.

The proposed tagger for information tag maintenance provideautomaticinformationtagsmaintenance,butitneeds manually dened tagging rules. This will be enough if we have complete control over information tag space, but the information tag space has been proposed to simplify sharing information among software systems [2] . Therefore third party software systems are still able to add, remove or modify information tags. In addition we have to maintain information tags created by users, too. To fulll this requirement we are proposing approach for automatic learning tagging rules based on analysis of developers activity streams and changes in information tag space. The approach is based on process discovery techniques [20]. This solution does not analyse modications of information tags but it gets new versions as combination of adding new information tags and removing old information tags. We associate information tags to streams of events about developers activities and analyse these streams by process discovery techniques. We hope, that this approach allows us to discover substantial software development processes that change properties of source code stored in information tags. To prove feasibility of this idea, we simplify process

[2]http://www.sonarqube.org/

[3]https://perconik.it.stuba.sk/codereview

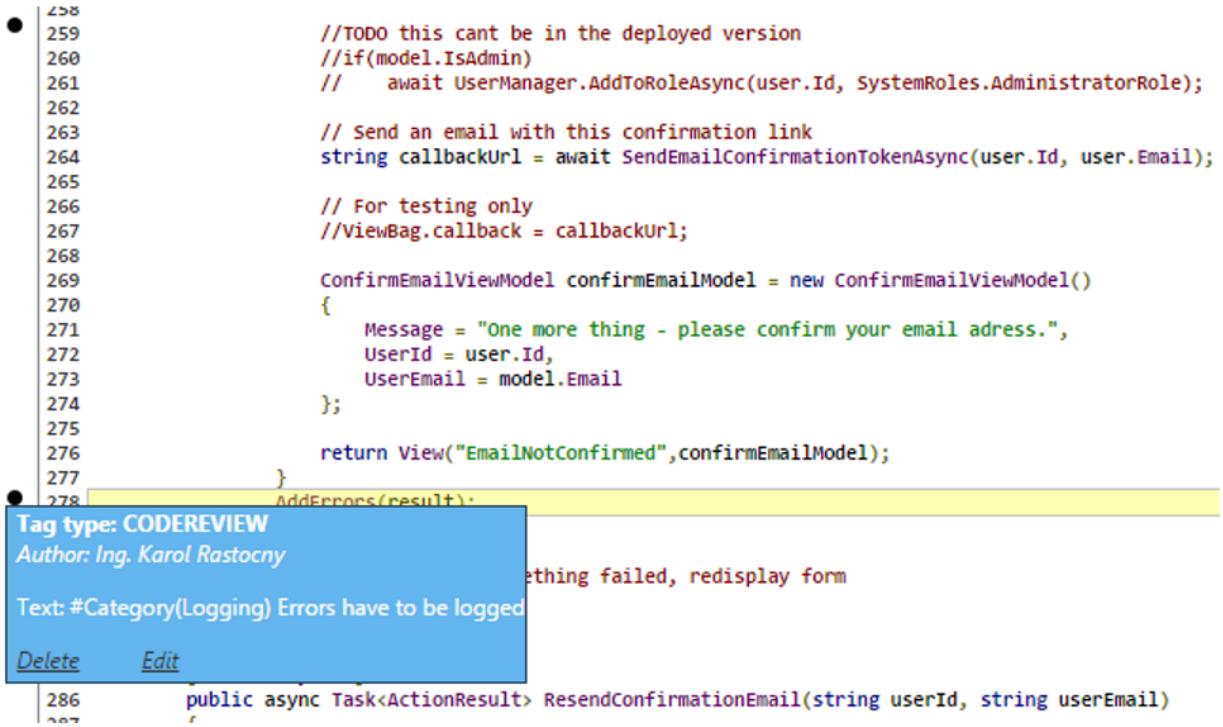

Fig. 3. Example of information tags in system CodeReview. There is ToDo information tag in the line 259. This tag has been written by developer during authoring source code. These code-written information tags are parsed after committing source code in to a repository. Information tag in line 278 is created by a reviewer in the system CodeReview during review process.

discovery to identication ofprocessesthatleadtobugsinsource-code.Inthissetupwe achieve promising preliminary results with precision above 70% and recall above 65%.

ACKNOWLEDGMENT

This work was partially supported by grants No. VG 1/0752/14 and No. VG 1/0646/15 and is the partial result of the Research & 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] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, and Michael Grossniklaus. An execution environment for c-sparql queries. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 441–452, New York, NY, USA, 2010. ACM.

[2] Mária Bieliková and Karol Rástočný. *Lightweight Semantics over Web Information Systems Content Employing Knowledge Tags*, pages 327–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[3] Raymond PL Buse and Thomas Zimmermann. Information needs for software development analytics. In *Proceedings of the 34th international conference on software engineering*, pages 987–996. IEEE Press, 2012.

[4] Luis Corral, Alberto Sillitti, Giancarlo Succi, Juri Strumpflohner, and Jelena Vlasenko. *DroidSense: A Mobile Tool to Analyze Software Development Processes by Measuring Team Proximity*, pages 17–33. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[5] Fausto Giunchiglia and Ilya Zaihrayeu. Lightweight ontologies. In *Encyclopedia of Database Systems*, pages 1613–1619. Springer US, 2009.