Should I Bug You? Identifying Domain Experts in Software Projects Using Code Complexity Metrics

Ralf Teusner, Christoph Matthies, Philipp Giese
Hasso Plattner Institute, University of Potsdam
August-Bebel-Str. 88
Potsdam, Germany
{firstname.lastname}@hpi.de

Abstract—In any sufficiently complex software system there are experts, having a deeper understanding of parts of the system than others. However, it is not always clear who these experts are and which particular parts of the system they can provide help with. We propose a framework to elicit the expertise of developers and recommend experts by analyzing complexity measures over time. Furthermore, teams can detect those parts of the software for which currently no, or only few experts exist and take preventive actions to keep the collective code knowledge and ownership high. We employed the developed approach at a medium-sized company. The results were evaluated with a survey, comparing the perceived and the computed expertise of developers. We show that aggregated code metrics can be used to identify experts for different software components. The identified experts were rated as acceptable candidates by developers in over 90% of all cases.

Keywords-domain experts; expert identification; software metrics; software quality;

I. Introduction

The *Bus Number* was informally defined by Coplien as the amount of developers that "would have to be hit by a truck (or quit) before the project is incapacitated" [1], with the worst answer to the question being "one". Losing developers in a software projects is especially disruptive if they were experts for a part of the system and contributed to the bus number. The most common ownership model for code is that of *subsystem ownership* [2], in which an expert takes primary responsibility for one or more software components. However, in large and usually also distributed software projects it is often not clear who these experts are and which parts of the system they have deep knowledge in [3]–[5]. As such, when needing detailed knowledge of a subsystem, the expert needs to be found in a time-consuming, manual process, including possibly multiple referrals. Management has a vested interested in determining and possibly increasing the bus number, i.e. the amount of expert developers, in order to make the project more resilient. Uneven distributions of experts can additionally be an indicator of low collective code ownership, a core concept of Extreme Programming. It describes the idea of every programmer being able to improve any code anywhere in the system [6]. High levels of collective code ownership can help in ensuring that the overall design is based on technical decisions, rather than following Conway's Law[1] [7]. It thus helps to encourage developers to feel more responsible for the quality of the whole project [2], [8]. The Analyzr framework enables analyses on the expertise of developers for parts of the system based on proven code complexity measures. It is publicly available as open-source software on GitHub [2] under the MIT license.

II. Related Work

Related work for the identification of domain experts using code complexity measures can be found mainly in the areas of measuring and aggregating source code metrics, code ownership as well as alternative methods of expert identification techniques.

A. Source Code Metrics

Coleman et al. [9] evaluate different software metrics in regards to their suitability as software maintainability predictors. The authors settle on the McCabe complexity as well as a set of Halsteads metrics. They point out that their approach could help "maintainers guide their efforts". Clark et al. [10] explore the use of software metrics in the area of autonomous vehicles. The authors rely solely on the McCabe complexity as they point out that a correlation between code errors and a high complexity was found [11]. Nagappan et al. [12] employ the complexity alongside a set of object oriented metrics to ascertain software properties. They argue that there is no universal set of metrics and that metrics have to be chosen on a per-project basis.

B. Aggregation of Code Metrics

Vasilescu et al. [13], [14] point out the need for aggregating metrics as most of them are defined on a micro level, such as functions or classes. However, conclusions have be drawn on a component or system level. Mordal et al. [15] point out the deficiencies of using the average to aggregate metric scores. They introduce the *Squale* model, which allows effectively comparing different metric values by normalising them into a given interval of values.

---

[1]Conway's Law states that the software structure developed in organizations reflects the organizational communication structure.

[2]https://github.com/frontendphil/analyzr

C. Code Ownership

Code ownership describes the approach of assigning source code or entire software systems to their human owners. these models can range all the way from a single *product specialist*, managing all the code to *collective ownership*, where responsibilities are shared amongst all developers [2].

Avelino et al. [16] propose an automated approach to estimating the *Truck Factor* (TF) of a project, a measure in the agile community of how prepared for developer turnover a project is. The authors state that the majority (65%) of the 133 surveyed systems extracted from GitHub had a TF ≤ 2.

Bird et al. [17] show that high levels of ownership were associated with less defects in the context of the two software products Windows Vista and Windows 7. Foucault et al. [18] attempted to replicate Bird's study in the context of free/libre and open-source software projects (FLOSS). They explored the relationship between ownership metrics and module faults in seven FLOSS projects, but only found a weak correlation. Thus, the authors conclude that the results of ownership studies performed using closed-source projects, which showed ownership metrics as accurate indicators of software quality, do not generalize to FLOSS projects.

Thongtanunam et al. [19] suggest complementing code ownership heuristics that rely on file authorship with code review metrics. This also includes those developers that contributed to the code by critiquing changes and suggesting edits.

D. Expert Identification

McDonald and Ackermann [20] describe a general architecture for *expertise locating systems*. They point out that these systems are not designed to replace key operational roles, such as a senior employee or guru, but can decrease workload and support decisions where previously no help was available. Furthermore, the authors state that organizationally relevant sources of information and heuristics need to be fitted to the work setting. They conclude that recommendation systems can help in finding experts who may not otherwise have been identified.

Schuler et al. [21] present an approach for retrieving the expertise of developers through analysis of method changes as well as method calls based on data gathered from code repositories. However, the authors do not take into account metrics that would indicate the quality of the code being examined.

Anvik et al. [22] present an approach to recommending a set of developers suited for assignment to bug tickets in the context of the *Mozilla* and *GCC* projects. They employed support vector machine classifiers, based on the one-line summary and full text description of collected bug reports that developers had previously been assigned to or had resolved. The feature vector was based on the frequency of terms in the text. The authors claim a precision of 64% precision for the *Firefox* project.

In the same problem domain, Tian et al. [23] propose a model for assigning developers to bug reports. Their model combines activity-based (developers who fixed similar bugs

in the past) as well as location-based techniques. The authors report that the most important similarity feature in their unified model was whether a developer had previously edited a file containing a potential bug. The proposed framework expands on this idea by enabling the rating of changes based on complexity metrics.

Venkataramani et al. [24] built a model of developer expertise in a target domain by mining developers activities in different open source projects. The example used is a recommendation system for StackOverflow based on data mined from GitHub. The system is based on *author/technical term* mappings extracted from source code and commits by authors in a bag-of-words model. In a subsequent step, technical terms associated with autors are mapped to StackOverflow tags. Unfortunately. the details on this mapping are not presented. The authors state that for a sample of 15 developers, 7 of them answered StackOverflow questions with tags that the model had discovered she was proficient in.

LaTozza et al. [25] point out that expertise not only means knowing more than others but also knowing where to look for the answer, or whom to ask. Their research revealed that interruptions by colleagues were ranked second when it came to issues hindering developers from working. Therefore approaches that identify more specific component experts and thereby spread the workload, would alleviate this burden for current experts.

III. COMPLEXITY MEASURES

The selection of appropriate complexity measures to determine developer expertise is vital to the quality of Analyzr results. Code metrics have to be chosen on a case by case basis as no single set of metrics can fit all use cases and contexts. As a basis for selecting metrics we propose Kaner's "Ten Measurement Factors" [26] for software metrics as well as further literature concerning the most relevant metrics for software design [27]–[30]. We employed metrics that measure independent aspects, used different approaches and incorporated different code parts in order to compute their results [31].

Figure 1 shows a summary of the development of a selection of code metrics at the company under study. In the shown timeframe, the company transitioned from a "start-up" phase, where the focus lay on fast feature introduction to support first customers, to a "sustainable" phase, focusing on a maintainable code base. The shift of focus is apparent in the increase of code quality around the end of 2011, where several refactorings took place. Since then, the Cyclomatic Complexity as well as the Halstead Volume have slowly begun to degrade again, as new code was added. This shows that real world circumstances are directly reflected in source code metrics, allowing insights into the development process.

In our study, the following three code complexity measurements were employed:

A. McCabe Complexity

The McCabe or *cyclomatic* complexity measure is derived from the amount of possible control flows that exist in a
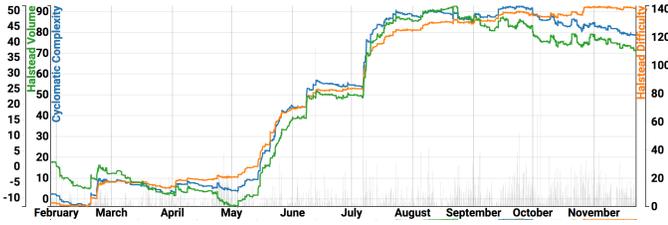
Fig. 1. Excerpt of changes in Halstead Volume (green) and Difficulty (orange) as well as Cyclomatic Complexity (blue) in the back end of the studied company. Deltas were oriented to indicate improved code quality, e.g. lower complexity, with rising chart lines. For brevity all charts were combined, however, as the deltas are from different domains, the absolute values do not allow direct comparisons. The grey bars indicate the amount of commits.

program [32]. A low McCabe complexity would thus be computed for a method with no branches and or one with only a simple type check in them. The metric operates on an abstract syntax tree and does not rely on a specific programming language [11]. The cyclomatic complexity v of a program G is defined as:

$$v(G) = e - n + 2p$$

where e is the number of edges, n is the number of nodes, and p is number of exit nodes (amount of possible program flow exits).

B. Halstead Metrics

Halstead introduced a set of metrics based on the concept that the complexity of a program will increase with the addition of new operators and operands [33]. Halstead based his metrics on four key variables: η_1, N_1, the number of distinct and total operators, as well as η_2, N_2 the number of distinct and total operands. We focus on *difficulty* and *volume*, as these are the most accepted in literature [9], [34]. They are defined as:

- *Volume*: $V = N \times \log_2 \eta$
- *Difficulty*: $D = \dfrac{\eta_1}{2} \times \dfrac{N_2}{\eta_2}$

C. Coupling

Coupling metrics describe the amount of dependencies between classes in a package and those outside of it. Software is easier to understand and maintain if it is split into modules of reasonable size [35]. However, coupling metrics are only applicable to programming languages that support object-oriented idioms such as classes and imports. While this is the case with Java, JavaScript does not natively support these concepts [36]. There are two types of coupling:

- *Efferent coupling*, also known as Fan-Out,
- *Afferent coupling*, also known as Fan-In.

Efferent coupling describes the number of classes that a given class depends on. Therefore the class can be affected by changes made elsewhere. While a high value in this metric does not necessarily represent bad design, it is often an indicator that the class has too many responsibilities, is poor in maintainability and should be split [37]. Afferent coupling

Fig. 2. Analyzr screenshot showing the changes in complexity measures for a single commit of the *Firebug* project.

Fig. 3. Analyzr screenshot showing developers ranked by expertise for a component of the *Firebug* package.

describes the amount of classes that depend on a given class. Changes in that class will affect all classes which depend on it. High values in either case of these metrics can be indicators for problems. Classes which have high values for both efferent and afferent coupling are often a source of bugs [12].

IV. THE ANALYZR FRAMEWORK

Analyzr identifies component domain experts by aggregating the results of various code complexity measurements on collected development data. Figure 2 shows the changes in metrics for a single commit[3]. Using this data, component domain experts are identified, see Figure 3.

Analyzr abstracts from the different version control systems that an organisation uses in order to allow analyses on a unified view of the repositories. For every repository, commit information, such as modified files and author, as well as meta-information about the repository itself, such as a list of branches, are gathered. This collected data is then used as input for proven third-party tools, specialized for the employed programming languages, which perform the chosen complexity measurements. The results of these tools are extracted, transformed into a common data model and saved in a typical *Extract, Transform, Load* (ETL) process, allowing analyses on well defined data structures. Analyzr aggregates the analysis results and presents visualizations to the user in a web interface, see Figure 1.

A. Architecture

Analyzr itself is split into a back end, using the Python web-framework Django [38], and a front end, using HTML5 and JavaScript, which is accessed with a browser. This allows

---

[3]https://github.com/firebug/firebug/commit/
076da997e6bc0cb14b27afcc2d845c730de52fcf
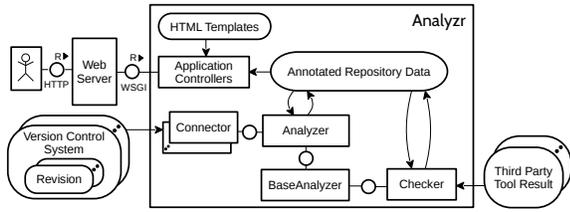
Fig. 4. FMC block diagram of the back end architecture of Analyzr.
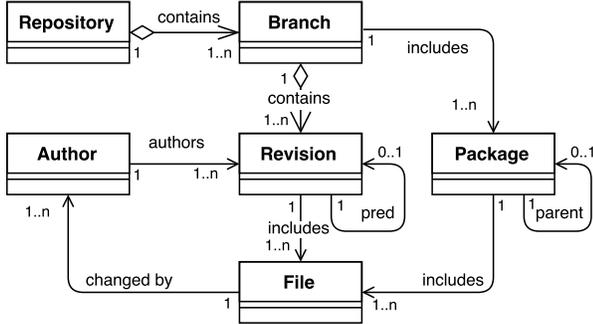
Fig. 5. UML class diagram of the models used to store gathered repository information.

Fig. 6. The interfaces which have to be implemented when adding a new *Connector* or *Checker* to Analyzr.

long-running analyses tasks to be performed on the server and not strain client resources. Figure 4 depicts the back end, which collects data from the different repositories that are to be analysed and stores the data which is produced during analyses. It exposes a REST interface to the front end, which presents a user interface to explore the data.

B. Data Model

The employed data model reflects the basic structure of a software repository. It is shown in Figure 5. A *Repo* entity holds information such as the location of the remote repository and user credentials and has a number of branches, which in turn are connected to a number of revisions, e.g. commits. For the sake of query performance and join avoidance, some information, such as the revision author is kept redundantly, in both the file and revision entities.

C. Extensibility

The task of communicating with the version control system is handled by *Connector* implementations. Currently, Subversion and Git repositories are supported. Every specialised connector is able to extract information from each revision in a given branch. Other version control systems can be added by implementing the minimal interface shown in Figure 6a. *Checker* classes (see Figure 6b) wrap third party code analysis tools, ensuring that they expose a common interface. Multiple checker instances can be used for analysing a certain programming language, so that that weaknesses in the analysis of one tool can be covered by another.

D. Third Party Tools

Our approach relies on proven, time-tested third party tools, that implement the described code metric algorithms. For Java as well as JavaScript, specialized tools were chosen.

1) JHawk: Java, as a statically typed language, allows computing a variety of metrics, ranging from generic complexity to object oriented ones [39]. JHawk [40] offers support for all of the software metrics required by Analyzr. It is possible to start measurements in a given directory and restrict the set of files which will be analysed. Using this mechanism we could incorporate the knowledge available in the version control system to only reassess changed files. JHawk produces an XML that is then further processed and loaded into the database.

2) Complexity Report: Complexity Report [41] is an open source JavaScript software, which can be run from the command line. As JavaScript is loosely typed, it does not natively facilitate the concept of packages, therefore C_e and C_a cannot be measured. Nevertheless, we are able to measure the McCabe complexity, the Halstead metrics, and the source lines of code (SLOC) using Complexity Report.

V. Metric Aggregation and Expertise Extraction

One of the main functions of Analyzr is to aggregate the different collected complexity measures into a single score, representing developer expertise for components. Metrics are computed on a file basis, instead of directly at package or component level, as this allows more fine-grained analyses and supports applying weighting operations earlier in the calculation. These values are then again aggregated on a directory level to form overall metrics for packages. However, some of the employed complexity measurements produce results on a function level. These low-level results need to be aggregated into a single value. Regular aggregation methods, such as mean and median are not suitable for all code metrics [14]. As code metrics have their own domain and semantics they are hard to compare. This is an issue Mordal et al. tried to solve with their *Squale* model [15].

A. Software Quality Enhancement (Squale)

Squale summarizes metric values by highlighting those which are problematic and weighs those values more that have recently improved [15], e.g. it emphasizes improvements in badly-rated system parts. In general, Squale provides a bounded, continuous scale for the comparison of metric values. It combines low-level marks into individual marks, which are then aggregated to a global mark:

The lower and upper thresholds describe those input values below and above which constants are returned for IM_i.

Metric	Formula	Lower Thr.	Upper Thr.	Ref.
Cyclomatic Complexity	$IM_{cc} = 2^{(7-cc)/3.5}$	2	19	[42]
Halstead Volume	$IM_{hv} = 3 - \frac{3 \times hv}{1000}$	20	1000	[43]
Halstead Difficulty	$IM_{hd} = 3 - \frac{3 \times hd}{50}$	10	50	[43]
Afferent Coupling	$IM_{C_a} = 2^{(30-C_a)/7}$	19	60	[42]
Efferent Coupling	$IM_{C_e} = 2^{(10-C_e)/2}$	6	19	[42]

1) **Low-level marks** are raw values retrieved from source code analysis, either manual metrics, assessed by humans, or automated tools, such as code metrics or rule checking.

2) **Individual marks** are computed from low-level marks. The thresholds for "good" or "bad" values with respect to project quality are determined by experts for the given field. They are mapped to a unified scale from 0 to 3 to allow comparisons.

After each individual mark (IM) has been computed they are aggregated using a weighting function. It is is defined as:

$$weight(\text{IM}) = \lambda^{-\text{IM}}$$

λ defines the strength of the weighting. Common strength values are 3, 9, and 30 for soft, medium, and hard, respectively. Hard weightings give more weight to bad results than soft weightings. The global mark GM^λ, combining all individual marks [15], is computed as:

$$\text{GM}^\lambda = -\log_\lambda \left(\frac{1}{|IM|} \cdot \sum_{i=1}^{|IM|} weight(\text{IM}_i) \right)$$

B. Computation of Employed Metrics

To compute the individual marks for the employed metrics presented in Section III we used the formulas presented in Table I, originally developed by Balmas et al. [42]. The *lower threshold* and *upper threshold* describe the boundaries above and below which constant values of 0 and 3 are returned. If the raw metric value is lower than the lower threshold, the code is assumed to not be complex and an individual mark of 3 is returned, indicating low relevance for refactoring. In the case of a raw metric value being larger than the upper threshold, an individual mark of 0 is returned, indicating a strong need for further review. Individual marks for raw metric values between the two thresholds are computed using the presented formulas in Table I.

C. Extracting Expertise of Developers

Our main goal is to determine the expertise of individual developers for specific program parts. For that purpose, we only regard developers having committed to the respective parts within a certain time frame[4]. Within this time frame, the set of revisions (r) created by an author (a), from the set of all authors (A) up to a certain point in time (T) is defined as:

$$R_T(a) = \{r \mid r \in R_T \wedge a \in A \wedge \text{author}(r) = a\}$$

Naive approaches like counting the number of commits or the number of lines changed as an indicator for expertise can be used for a first estimation. However, we are confident that for developers in an established team the expertise concerning a program part is better reflected in the quality of the code they produce. As most development takes place on an already existing code base, we focus on the changes of the quality metrics, i.e. deltas, in code metrics. For each revision within the set $R_T(a)$, we check whether the global mark, computed using Squale, improved or deteriorated. These changes are attributed to the developer who authored the commit. We count all commits resulting in increases and decreases of software quality and calculate the developer's individual score ($\text{score}_T(a)$):

$$qi = min \left(\frac{\text{increases}_T(a)}{\text{decreases}_T(a)}, 1 \right)$$
$$\text{score}_T(a) = qi \times \ln \left(1 + |R_T(a)| \right)$$

The ratio of commits that increase versus those that decrease software quality (qi, quality impact) is multiplied with the logarithmic smoothed amount of total commits of the author. The formula incorporates the total amount of commits, but does not double the score for double the number of commits, which would skew the score in favor of long-serving developers. In the unlikely case that a developer has not produced a commit which decreased the code quality, we fall back to evaluating the number of commits that were produced, as qi is 1. However, the normal case in software development is that there are more commits which decrease, rather than increase code quality [44], resulting in qi being lower than 1.

The score therefore combines the experience of a developer, expressed by the number of his commits with his quality impact, to reflect his expertise.

VI. TEST BED SELECTION

In order to evaluate the quality of Analyzr results, the software was deployed at a collaborating software company and real development data was analyzed. The company chosen was a German software development company in the sector of business process modelling. It was chosen as one of the authors was employed there and had insights into the structures and processes within the company.

[4]In our study we picked 62 days, as this served as a distinguishing factor between temporary and permanent leave, in the context of the studied company.

A. Company Introduction

As the company that was analyzed builds mostly applications for the web, they practice a separation between front and back end code. Of the then about thirty developers in the company, around ten are active in the front end and the other twenty are concerned with back end development. The back end of the software is implemented in Java, whereas the front end is a JavaScript application. Furthermore, the developed software is split into well-defined components, providing clearly bound domains of expertise. Currently code metrics are not an official part of the development process of the company. However, code reviews are performed amongst developers, with the aim of keeping the code maintainable.

B. Time Frame

When attempting to determine the current experts of a software project the history of the project that is being analyzed needs to be restricted to allow focusing on those developers that have recently contributed code and are well-versed with the current status of the software. For the studied company, our observations and interviews pointed to an optimal time frame for analysis of around 62 days. This period is long enough so developers who are on a temporal leave do not get sorted out, but it is short enough to exclude developers who are no longer active in a certain module or have left the company.

C. Threats to Validity

Real world development data from a company has inconsistencies, that make analysis challenging. In the data set collected for this paper, credentials of developers differed over time, for example if a developer changed his user name in the version control system. We consolidated these changes in a manual correction phase. Data was gathered from the *master* (Git) and *trunk* (SVN) branches of the version control systems, as only this code represents usable increments of the software. Code that was in other, possibly specialized development or feature branches was not analyzed.

VII. EVALUATION

In order to evaluate the results of Analyzr and compare them to the expectations of the developers, a survey was devised. Survey participants were developers who volunteered. Overall, 13 developers participated, aged 20 to 30 years. Among them were 12 males and 1 female; 5 participants were seniors, while 8 were junior developers. For the following analysis all names were anonymized. The survey consisted of two main parts:

- **Expert Selection** Developers self-assessed whether they were active in the front or the back end. Based on their decision they were presented with the respective components of their division and were asked to select an expert for each. They were free to name two other qualified developers as well.
- **Proposal Evaluation** Developers were presented with the top three component experts identified by Analyzr and were asked to rate the accuracy of each result.

FRONT END COMPONENTS

Component	1st choice	2nd choice	3rd choice
Administration	Marler (100%)	Prud (100%)	Braaten (100%)
Analytics	Marler (60%)	Braaten (100%)	Marler (100%)
Comparator	Marler (100%)	Waring (100%)	Marcuso (50%)
Editor	Marler (100%)	Moyer (60%)	Mayberry (67%)
Explorer	Marler (100%)	Waring (50%)	Mayberry (67%)
Glossary	Marler (100%)	Waring (100%)	Moyer (67%)
Utils	Moyer (60%)	Mayberry (50%)	Waring (100%)
Portal	Marler (100%)	Waring (100%)	Moyer (100%)
Quick Model	Marler (60%)	Waring (100%)	Marler (50%)
Simulation	Marler (40%)	Braaten (50%)	Marler (100%)
Testing	Moyer (80%)	Salmeron (50%)	Marler (25%)

BACK END COMPONENTS

Component	1st choice	2nd choice	3rd choice
Diagram API	Marston (88%)	Gillette (67%)	Prouty (50%)
Glossary	Anstine (38%)	Marston (57%)	Gillette (33%)
Platform	Anstine (100%)	Prud (57%)	Marston (50%)
SVG Renderer	Marston (88%)	Gillette (50%)	Braaten (50%)
User Mgmt	Anstine (88%)	Prud (50%)	Braaten (40%)
Warehouse	Anstine (100%)	Prud (75%)	Braaten (33%)

TABLE II
EXPERTS FOR THE DIFFERENT FRONT END (TOP) AND BACK END (BOTTOM) COMPONENTS AS VOTED BY COMPANY DEVELOPERS. DISPLAYED PERCENTAGES REPRESENT THE CONSENSUS OF DEVELOPERS FOR A GIVEN RANK.

A. Expert Selection

Table II summarizes the selections of the company's staff for first, second and third choice of rank of expertize for each software component. The percentages given represent the consensus of the developers for the given rank, meaning that for example Marler was chosen most often as the best expert for Analytics (representing 60% of the votes), and the missing 40% of the vote-shares for the first rank favored other candidates. A developer could be nominated as first, second and third choice at the same time, if respondents see someone as the only choice (or best choice and last fallback, if the second best expert is not available). For 6 components out of 11 in the front end and 2 components out of 6 in the back end all surveyed developers agreed on the first choice of expert. High uncertainty among developers concerning expert selection, i.e. less than 50% agreement, was only apparent in 6 out of 51 choices. This data lends credibility to the hypothesis that developers have a specific component expert in mind, whom they would most likely consult if they had a question in that specific domain. Furthermore, we found a high level of agreement who was considered to be among the top experts when asking the developers. However, both in the front as well as in the back end, only two distinct developers were voted as first choice: Marler and Moyer as well as Marston And Anstine, respectively. While this is certainly flattering for these developers, it also introduces problems. They will most likely be interrupted frequently and if they are out of the office, developer's single point of contact in case of questions is lost.

The accuracy of Analyzr results was evaluated by comparing the level of agreement between the lists of manually

and automatically selected component experts. The developers, who chose the experts manually, did not have prior knowledge of Analyzr results. Figure 7 summarizes the accuracy of Analyzr predictions for the first choice of component experts for the whole company repository. If the prediction equalled the manually identified expert, it was classified as a match. Otherwise an Anaylzr result can be classified one or two off, depending on whether the prediction missed the top result by one or two ranks. If the prediction was not included in the list of manually selected component experts it was considered a miss. Analyzr was able to produce a match in 47,37% of observed cases, with 15,79% of cases classified as one or two ranks off. This is influenced by the fact that in the front end in 50% of cases the manually identified expert was missed. This raises the question whether some of the developers simply did not know who the component experts were and misidentified them, or whether Analyzr results were inaccurate. In order to answer this question the expert proposals by Analyzr were directly rated by the developers.

B. Proposal Evaluation

For each of the three experts identified by Analyzr for the software components, all interviewed developers were asked to rate the results on a 4-point scale: "strongly disagree" (0), "disagree" (1), "agree" (2), "strongly agree" (3). The choice of "neutral" was deliberately omitted, as it has no semantics concerning the acceptance of a proposed expert. Considering the evaluated expertise of a developer, this means the proposed developer ...

0 ... does not have knowledge of the component.

1 ... does not have enough knowledge to be considered a component expert.

2 ... is acceptable as component expert, but someone else would be better suited.

3 ... is identified correctly as component expert and the result is considered useful.

The ratings regarding the first choice of expert presented by Analyzr are summarized in Figure 8. Overall, experts recommended by Analyzr as the first choice were rated as either acceptable or completely correct (2 or 3 points) with a rate of 100% in the back end and 90% in the front end. This is evidence for the hypothesis that there are non-obvious component experts that are unknown to developers and cannot be identified by simply asking staff members. However, Analyzr was able to identify these experts who were accepted by the vast majority of survey participants.

VIII. CONCLUSION AND FUTURE WORK

In this paper we presented Analyzr, a framework for eliciting the expertise of developers for software components using complexity analysis on source code. We showed that using our approach, it is feasible to extract component experts in a case study with a medium-size software development company. The experts we found and suggested differed from those that developers picked intuitively. However, the algorithmically extracted experts were rated by developers as accurate in the vast majority of cases. This lends credibility to the hypothesis that Analyzr was able to find "hidden experts", i.e. developers who have a lot of specific component knowledge and can answer questions, but may not be the intuitive choice. Identifying these experts is helpful for organisations as no longer all questions and enquiries have to be directed to the single obvious expert. This relieves pressure from this expert and enables wider distribution of knowledge as more developers become involved in solving challenging questions. Furthermore, developers can visualize and track their statistics, see the progress they are making, which provides an additional incentive to produce cleaner code. Future work will focus on iteratively improving the software based on user feedback after having productively used it at the surveyed company for an extended period of time. Especially, improvements to metric selection and thresholds will further enhance the accuracy of results. For the case of analysing object-oriented languages, more specific metrics, such as the Method Hiding Factor or the Attribute Hiding Factor, could be employed [39]. These can provide better insights into the code base at the cost of general applicability.

REFERENCES

[1] L. Williams and R. Kessler, *Pair Programming Illuminated*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[2] M. E. Nordberg III, "Managing code ownership," *IEEE Softw.*, vol. 20, no. 2, pp. 26–33, Mar. 2003.

[3] K. Ehrlich and K. Chang, "Leveraging expertise in global software teams: Going outside boundaries," in *Proceedings of the IEEE International Conference on Global Software Engineering*, ser. ICGSE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 149–158.

[4] S. Faraj and L. Sproull, "Coordinating expertise in software development teams," *Manage. Sci.*, vol. 46, no. 12, pp. 1554–1568, Dec. 2000.

[5] J. Herbsleb and R. Grinter, "Architectures, coordination, and distance: Conway's law and beyond," *Software, IEEE*, vol. 16, no. 5, pp. 63 –70, sep/oct 1999.

[6] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

[7] M. E. Conway, "How do committees invent," *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[8] Agile Alliance, "Agile glossary — collective ownership," 2015, [accessed 4-August-2016]. [Online]. Available: https://www.agilealliance.org/glossary/collective-ownership/

[9] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer*, vol. 27, no. 8, pp. 44–49, 1994.

[10] M. Clark, B. Salesky, C. Urmson, and D. Brenneman, "Measuring software complexity to target risky modules in autonomous vehicle systems," in *Proceedings of the AUVSI North America Conference*, 2008.

[11] A. H. Watson, T. J. McCabe, and D. R. Wallace, "Structured testing: A testing methodology using the cyclomatic complexity metric," *NIST special Publication*, vol. 500, no. 235, pp. 1–114, 1996.

[12] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 452–461.

[13] B. Vasilescu, A. Serebrenik, and M. van den Brand, "By no means: A study on aggregating software metrics," in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics*. ACM, 2011, pp. 23–26.

[14] ——, "You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Sept 2011, pp. 313–322.

[15] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, and S. Ducasse, "Software quality metrics aggregation in industry," *Journal of Software: Evolution and Process*, 2012.
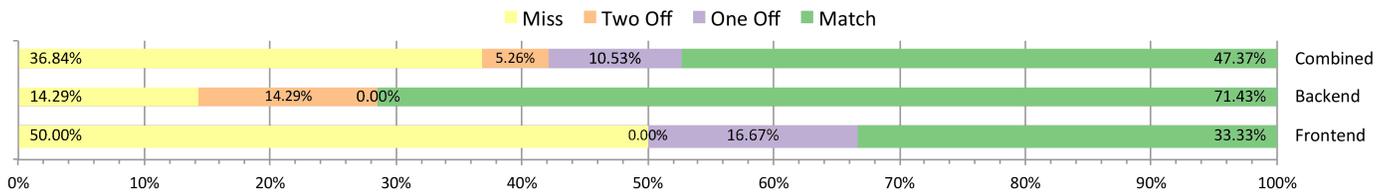
Fig. 7. Accuracy of our findings considering only the first choice compared to the statements made by the staff. The results are grouped into perfect match, one off, two off, and miss. The combined percentages are skewed towards the front end, as there are more front than back end components.
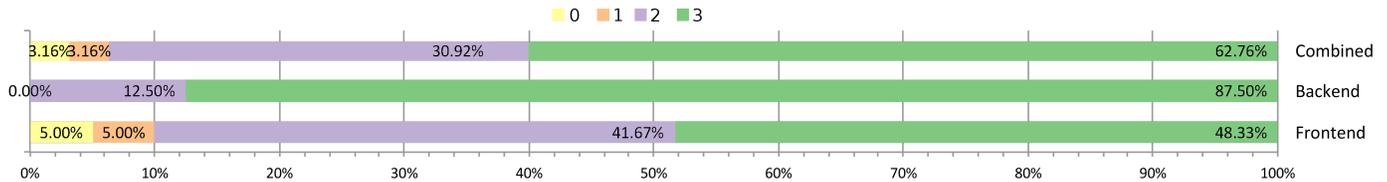
Fig. 8. Acceptance of the computed experts, regarding only the first choice by the staff. The numeric values represent: disagreement (0), weak disagreement (1), weak agreement (2), agreement (3). The combined percentages are skewed towards the front end, as there are more front than back end components.

[16] G. Avelino, L. Passos, A. Hora, and M. T. Valente, "A novel approach for estimating Truck Factors," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, vol. 2016-July, no. Dcc. IEEE, may 2016, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/document/7503718/

[17] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't Touch My Code!: Examining the Effects of Ownership on Software Quality," *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 4–14, 2011.

[18] M. Foucault, J.-R. Falleri, and X. Blanc, "Code ownership in open-source software," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14*. New York, New York, USA: ACM Press, 2014, pp. 1–9. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2601248.2601283

[19] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Revisiting code ownership and its relationship with software quality in the scope of modern code review," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, no. 1. New York, New York, USA: ACM Press, 2016, pp. 1039–1050. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2884781.2884852

[20] D. W. McDonald and M. S. Ackerman, "Expertise Recommender: A Flexible Recommendation System and Architecture," in *Proceedings of the 2000 ACM conference on Computer supported cooperative work - CSCW '00*. New York, New York, USA: ACM Press, 2000, pp. 231–240. [Online]. Available: http://portal.acm.org/citation.cfm?doid=358916.358994

[21] D. Schuler and T. Zimmermann, "Mining usage expertise from version archives," in *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 2008, pp. 121–124.

[22] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134336

[23] Y. Tian, D. Wijedasa, D. Lo, and C. Le Goues, "Learning to rank for bug report assignee recommendation," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, vol. 2016-July. IEEE, may 2016, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/document/7503715/

[24] R. Venkataramani, A. Gupta, A. Asadullah, B. Muddu, and V. Bhat, "Discovery of technical expertise from open source code repositories," in *Proceedings of the 22Nd International Conference on World Wide Web*, ser. WWW '13 Companion. New York, NY, USA: ACM, 2013, pp. 97–98. [Online]. Available: http://doi.acm.org/10.1145/2487788.2487832

[25] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 492–501.

[26] D. Hoffman, "The darker side of metrics," in *Pacific Northwest Software Quality Conference, Portland, Oregon*, 2000.

[27] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 2009, pp. 367–377.

[28] A. German, "Software static code analysis lessons learned," *Crosstalk*, vol. 16, no. 11, 2003.

[29] C. Kaner and W. P. Bond, "Software engineering metrics: What do they measure and how do we know?" *methodology*, vol. 8, p. 6, 2004.

[30] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *Software Engineering, IEEE Transactions on*, vol. 22, no. 10, pp. 751–761, 1996.

[31] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *Software Engineering, IEEE Transactions on*, vol. 26, no. 7, pp. 653–661, 2000.

[32] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[33] M. Halstead, "Potential impacts of software science on software life cycle management," 1977.

[34] V. Y. Shen, S. D. Conte, and H. E. Dunsmore, "Software science revisited: A critical analysis of the theory and its empirical support," *Software Engineering, IEEE Transactions on*, no. 2, pp. 155–165, 1983.

[35] S. Henry and D. Kafura, "Software structure metrics based on information flow," *Software Engineering, IEEE Transactions on*, no. 5, pp. 510–518, 1981.

[36] D. Crockford, *JavaScript: the good parts*. O'Reilly Media, Inc., 2008.

[37] R. M. Redin, M. F. Oliveira, L. B. Brisolara, J. C. Mattos, L. C. Lamb, F. R. Wagner, and L. Carro, "On the use of software quality metrics to improve physical properties of embedded systems," in *Distributed Embedded Systems: Design, Middleware and Resources*. Springer, 2008, pp. 101–110.

[38] Django Software Foundation, "Django web-framework," https://www.djangoproject.com/.

[39] R. Harrison, S. Counsell, and R. Nithi, "An overview of object-oriented design metrics," in *Software Technology and Engineering Practice, 1997. Proceedings., Eighth IEEE International Workshop on [incorporating Computer Aided Software Engineering]*. IEEE, 1997, pp. 230–235.

[40] Virtual Machinery, "Jhawk product overview," 2015, [accessed 4-August-2016]. [Online]. Available: http://www.virtualmachinery.com/jhawkprod.htm

[41] escomplex, "complexity-report," 2016. [Online]. Available: https://github.com/escomplex/complexity-report

[42] F. Balmas, F. Bellingrad, F. Denier, S. Ducasse, B. Franchet, J. Laval, K. Mordal-Manet, and P. Vaillergues, "The squale quality model. modèle enrichi d'agrégation des pratiques pour java et c++," INRIA, Tech. Rep., 2010.

[43] A. Serebrenik, "2is55 software evolution," 2011.

[44] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.