Exploring Crowdsourced Reverse Engineering

Sebastian Heil, Felix Förster and Martin Gaedke
Technische Universität Chemnitz, 09107 Chemnitz, Germany

Keywords: Reverse Engineering, Crowdsourcing, Microtasking, Concept Assignment, Classification, Web Migration, Software Migration.

Abstract: While Crowdsourcing has been successfully applied in the field of Software Engineering, it is widely overseen in Reverse Engineering. In this paper we introduce the idea of Crowdsourced Reverse Engineering and identify the three major challenges: 1) automatic task extraction, 2) source code anonymization and 3) quality control and results aggregation. To illustrate Crowdsourced Reverse Engineering, we outline our approach for performing the Reverse Engineering activity of concept assignment as a crowdsourced classification task and address suitable methods and considerations with regard to each of the the three challenges. Following a brief overview on existing research in which we position our approach against related work, we report on our experiences from an experiment conducted on the crowdsourcing platform microworkers.com, which yielded 187 results by 34 crowd workers, classifying 10 code fragments with decent quality.

1 INTRODUCTION

Migration of legacy systems to the Web is an important challenge for companies which are developing software. Driven by the high number of ways in which users interact with recent web applications, changing user expectations pose new challenges for existing non-web software systems. Continuous evolution of technologies and the termination of support for obsolete technologies furthermore intensify the pressure to renew these systems (Wagner, 2014). As web browsers are becoming the standard interface for many applications, web applications provide a solution to platform-dependence and deployment issues (Aversano et al., 2001). Many companies are aware of these reasons for web migration. On the other hand, in particular Small and Medium-sized Enterprises (SMEs) find it difficult to commence a web migration (Heil and Gaedke, 2017).

Using LFA[1] problem trees, we identified *doubts about feasibility* as one of the main factors which are keeping SME-sized software developing companies from migrating their existing software products to the web. This is mainly due to the danger of losing knowledge. Successful software products of small and medium-sized software providers are often specifically tailored to a certain niche domain and result from years of requirements engineering (Rose et al., 2016). Thus, the amount of valuable domain knowledge from problem and solution domain (Marcus et al., 2004) such as models, processes, rules, algorithms etc. represented by the source code is vast. (Wagner, 2014) The redevelopment required due to the many paradigm shifts for web migration – client-server separation in the spatial and technological dimension, asynchronous request-response-based communication, explicitely addressable application states via URLs and navigation to name but a few – bear the risk of losing this knowledge.

However, in legacy systems, domain knowledge is only implicitly represented by the source code and often poorly documented (Warren, 2012; Wagner, 2014). *Reverse Engineering* is required to elicit the knowledge, make it explicit and available for subsequent web migration processes. Existing redocumentation approaches (Kazman et al., 2003) are not feasible for small and medium-sized enterprises since they cannot be integrated into day-to-day agile development activities. Therefore, we introduced an approach based on source code annotations (Heil and Gaedke, 2016) which allows to enrich the legacy source code by directly linking parts of it with representations of the knowledge which they contain. Supported by a web-based platform, this enables developers to reference the knowledge in emails, wikis, task descriptions etc. and to jump directly to their de-

[1]Logical Framework Approach, cf. http:// ec.europa.eu/ europeaid/

Heil, S., Förster, F. and Gaedke, M.
Exploring Crowdsourced Reverse Engineering.
DOI: 10.5220/0006758401470158
In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, pages 147-158
ISBN: 978-989-758-300-1

finition and location in the legacy source code. The identification of domain knowledge in source code is known as *concept assignment* (Biggerstaff et al., 1994).

While manual concept assignment can easily be integrated into the daily development activities of the small and medium-sized enterprise and allows to incrementally re-discover and document the valuable domain knowledge, it still requires a high amount of effort and time, in particular taking into account the limited resources of small and medium-sized enterprises. This process involves reading a source code, selecting a relevant area of it and determining the type of knowledge which this area represents, before further analysis can extract model representations of the knowledge. This can be considered a *classification task*. Crowdsourcing has a history of successful application in classification tasks. Also, crowdsourcing has been successfully employed in software engineering contexts, in particular on smaller tasks without interdependencies (Stol and Fitzgerald, 2014)(Mao et al., 2017). Thus, in this work we explore crowdsourced reverse engineering (CSRE) on the example of identification of knowledge type in legacy codebases.

Challenges of the Application of Crowdsourcing in Reverse Engineering include:

1. automatic extraction and preparation of crowdsourcing tasks from the legacy source,
2. balancing controlled disclosure of proprietary source code with readability and
3. quality control and aggregation of results.

In order to create suitable tasks for crowdsourcing platforms, the legacy source code has to be split into fragments which can then be classified by the crowd workers. These fragments should be large enough to provide sufficient context for a meaningful classification and small enough to allow for a unambiguous classification and a good recall. Since the legacy source is a valuable asset of the company, public disclosure of code fragments on a crowdsourcing platform needs to be well controlled. Competitors should not be able to identify the authoring company, the software product or the application domain, in order to prevent them from gaining insights on the software product or even replicating parts of it. However, the required anonymization needs to be balanced against the readability of the code. Code obfuscation algorithms produce results that are intendedly hard to read (Ceccato et al., 2014), which jeopardizes getting high quality classification results from the crowd. Controlling the quality and aggregating the classification results is a key challenge. In particular, effort needed to ensure a decent classification quality should not mitigate the advantage gained by crowdsourcing. Fake contributions should be filtered and contradicting classifications have to be aggregated.
In the following paper, we report on our experiences in the application of crowdsourcing in the reverse engineering domain. We briefly outline our approach in 2, detail the three aforementioned challenges of automatic task extraction in 3, source code anonymization in 4 and quality control and results aggregation in 5. We position our approach against existing work in 6, report on the results from a small-scale validation experiment in 7 and conclude the paper with an outlook on open issues in 8.

2 APPROACH

Figure 1 shows an overview of the crowdsourcing-based classification process for reverse engineering. There are three roles involved: *Migration Engineers* are the actors in charge of conducting the migration, the *Annotation Platform* is a system role representing our platform for supporting web migration through code annotation (Heil and Gaedke, 2016), and *CS Platform* represents a crowdsourcing platform allowing to post classification tasks to crowd workers.

A migration engineer starts the process by defining the scope on the legacy codebase. This scope can be defined in terms of selected source files, software components (represented by project/solution files) or the complete code base. Next, the annotation platform automatically extracts code fragments for classification as described in section 3.

The next step is the pre-processing of the extracted fragments for achieving the intended anonymization properties, which we describe in section 4.

Then, the annotation platform deploys classification tasks for each of the anonymized code fragments in the CS Platform . The data passed to the CS Platform includes a brief description of the reverse engineering classification task, a URL pointing to the crowdworker classification view (figure 2) in the annotation platform, the crowdworker requirements and the reward configuration. Crowdworkers matching the requirements are provided with a description of the categories for classification, following our ontology for knowledge in source code (Heil and Gaedke, 2016). The URL pointing to the crowd worker view can either be presented as a link or integrated in the CS platform using an iframe, depending on the technological capibilities and terms of use of the CS platform.

In the crowd worker view, the crowd worker is presented with the code fragment to be classified, and

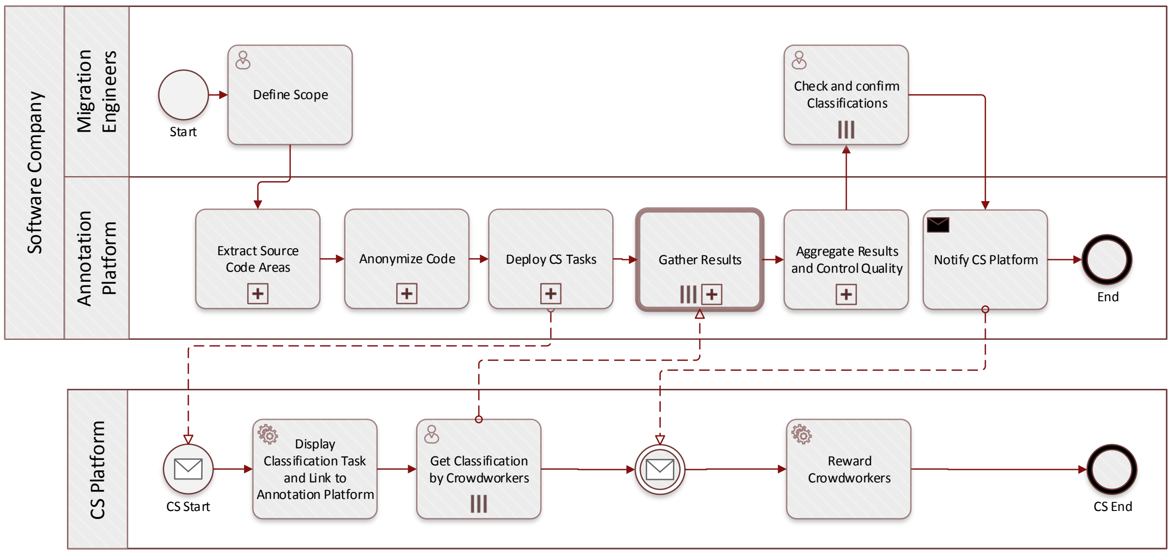
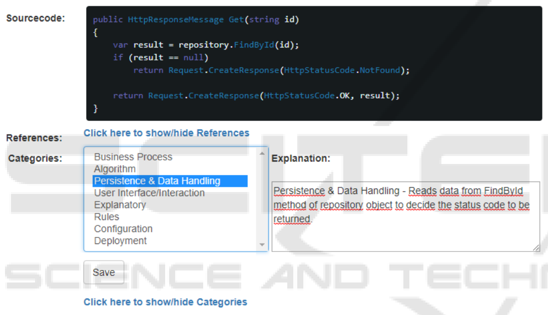

Software Company
Migration Engineers
Annotation Platform
Start
Define Scope
Check and confirm Classifications
Extract Source Code Areas
Anonymize Code
Deploy CS Tasks
Gather Results
Aggregate Results and Control Quality
Notify CS Platform
End
CS Platform
CS Start
Display Classification Task and Link to Annotation Platform
Get Classification by Crowdworkers
Reward Crowdworkers
CS End

Figure 1: Crowdsourcing-based source code classification process.


Sourcecode:

```
public HttpResponseMessage Get(string id)
{
    var result = repository.FindById(id);
    if (result == null)
        return Request.CreateResponse(HttpStatusCode.NotFound);

    return Request.CreateResponse(HttpStatusCode.OK, result);
}
```

References: Click here to show/hide References

Categories:
- Business Process
- Algorithm
- Persistence & Data Handling
- User Interface/Interaction
- Explanatory
- Rules
- Configuration
- Deployment

Explanation: Persistence & Data Handling - Reads data from FindById method of repository object to decide the status code to be returned.

Save

Click here to show/hide Categories

Figure 2: Crowd worker view.

the list of possible categories. The crowdworker view additionally contains a list of source code references, allowing the crowdworker to review also the source code of pieces of source code which are referenced. Basic authentication of crowd workers is achieved through user-specific urls and temporary tokens.

The classification results are then aggregated and quality control measures are applied as described in section 5. The filtered results can then be automatically included in the annotation platform, or they are marked for review by a migration engineer, who can accept or reject them. Ultimately, the annotation platform notifies the CS platform so that the participating crowd workers can be rewarded as configured.

In the following three sections, we provide details about the components addressing the main challenges raised in the introduction.

3 CLASSIFICATION TASK EXTRACTION

Micro-tasks are characterized as self-contained, simple, repetitive, short, requiring little time, cognitive effort and specialized skills (Stol and Fitzgerald, 2014). Of these properties, classification of code fragments matches the first five: classification results are not dependent on other classification results, the classification is a simple selection from a list of available classes, the classification activity is highly repetitive and a single classification can be achieved in relatively low time. Compared to other successfully crowdsourced classification tasks like image classification, a higher cognitive effort is required. Specialized skills are required, because the crowd workers have to have a sufficient reading understanding of the provided source code. However, since reading and understanding enough of a source code to determine the correct class requires only a basic understanding of programming and limited knowledge of the programming language used, the skill requirements are not too high. It is suitable for a wider range of crowd workers in comparison to crowdsourcing the development an application.

In order to extract micro classification tasks from the legacy code base, the source code has to be automatically divided. The code fragments for classification are identified by analysis of the source code structure. Suitable methods serving this purpose must have three essential **Classification Task Extraction Properties:**

1. Automation
2. Legacy language support
3. Completeness of references

Automation. A suitable extraction method should not require additional user interaction to perform the analysis and to carry out the identification of relevant code fragments for classification.

Legacy Language Support. Since source code analysis is programming language specific, the method should support the most common programming languages. According to IEEE SPEKTRUM[2], the ten most widely used programming languages are: C, Java, Python, C++, R, C#, PHP, JavaScript, Ruby and Go. While Go is a relatively new language (appeared in 2009) and Ruby and Javascript have only recently seen an increased use in the context of web applications, R is a language mainly used for statistics and data analysis. Typical languages found in legacy software to a larger extent include C, C++ and Java.

Completeness of References. For a crowd worker to have sufficient information to properly categorize a code fragment, he must be able to understand the control and data flow. To provide this information to the crowd worker, the extraction method must also provide information about source code which is referenced in the code fragment.

We analyzed three groups of approaches for classification task extraction. *Documentation tools* are originally used to automate the creation of source code documentation. Instead of developing specific extraction tools, existing documentation tools can be re-used: Since the structure of the source code is analyzed in order to create the documentation, this group of methods offer possibilities for identifying structural properties of source code. *Syntactic analysis tools* explicitely analyze code regarding its structural properties. There are two different types: regular-expression-based and parsers. Regular expressions are used to recognize patterns in texts. Thus, a set of regular expressions allows identifying relevant source code areas for classification. Parsers create representations of the syntactical structure of a program. Abstract syntax trees are used to represent the structure and sequence of program code. *Syntax highlighting tools* usually generate custom representations of source code structure in order provide syntax highlighting in text editors.

We systematically investigated the applicability of these three groups for the extraction of classification tasks against the three aforementioned essential properties as requirements. For most programming languages, production-grade implementations of documentation tools exist. Source code references are completely traceable, ensuring good understanding for crowd workers. Automation is easily achieved due to the capability to configure the extraction process by command line parameters. Regular expressions are a standardized means of extracting information from text and are supported by all current programming languages. Thus they can be employed for automatic extraction from within a surrounding custom extraction program. However, source code references can only be traced with high effort and with many iterations of using regular expressions. Parsers, on the other hand, allow the tracking of source code references by analyzing the data and control flow and also exist for most programming languages. However, the analysis results generated in the parsing process can either not be exported or are only available as graphical representations, making further processing difficult. Therefore, their use in an automated extraction process is significantly limited. While syntax highlighting tools allow the identification of code fragments and, create a structure overview internally, support for exporting the structure file is only available for certain platforms. As a result, their applicability is limited. Based on these considerations and a feasibility study by students, we decided to use documentation tools as basis for the fully automated classification task extraction. Our prototypical implementation employs the tool “Doxygen”[3] and parses the generated documentation to identify relevant code fragments.

4 SOURCE CODE ANONYMIZATION

In crowdsourcing, the crowd workers respond to an open call. They are unknown to the organization and the group of workers is potentially large.(Latoza and van der Hoek, 2016) Therefore, placing a task on a crowdsourcing platform implies making the task contents publicly available. This bears the risk that competitors could access the code fragments published in the crowd tasks and make unintended use of them.

To enable companies to employ CSRE, means of *source code anonymization* are therefore relevant. Code obfuscation techniques provide means to change source code to make it harder for human readers to understand maintaining the original functionality (Ceccato et al., 2014). While this would provide a certain protection from unintended distribution of a company’s valuable source code, it also severely im-

[2]http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages

[3]https://www.stack.nl/ dimitri/doxygen/

pacts the readability. The impact of code obfuscation techniques on understandability by human readers has been assessed in (Ceccato et al., 2014).

The challenge of source code anonymization in the context of CSRE is to balance information disclosure and readability. While a suitable anonymization method should modify a source code to sufficiently prevent unintended use, it has to maintain its readability so that crowd workers are still able to achieve a level of understanding of the code sufficient for performing the reverse engineering task.

This necessary balance is expressed in the following **Anonymization Properties**. A suitable anonymization method must:

1. Prevent identification of software provider, software product and application domain
2. Maintain the information relevant for classification and the control flow
3. Avoid negative impact on readability of the source

In the following, we address these three anonymization properties. Since achievement of any of the properties influences the others, we do not structure this section per property. Any obfuscation techniques which alters the syntactic sequence of expressions, for instance by code optimizations such as inline expansion[4] or by adding artificial branches to the control flow (cf. opaque predicates(Arboit, 2002)) is disregarded because the control flow is not maintained. In source code obfuscation, *identifier renaming* has shown good results (Ceccato et al., 2014). Identifiers, however, are not the only parts of code which contain information that allows identification of software provider, software product or application domain (referred to as *identification information* in the following). There are three different *loci of identification information*: identifiers, strings and comments. While code obfuscation has to produce identical software, for anonymization, modifications can also be applied to string contents, since the altered source code is only presented to crowd workers for reverse engineering and not used to compile to running software.

In contrast to code obfuscation, where identifier renaming typically yields intendedly meaningless random combinations of characters and numbers, replacements for source code anynomization in the context of CSRE have to maintain readability and information content as good as possible. The naïve approach would be to create custom lists of words to be replaced and mappings onto their respective replacements. However, this requires a high manual effort and the completeness of the anonymization highly depends on the completeness of theses lists. For larger code bases as found in the professional software production context of small and medium-sized enterprises as outlined in the introduction, this is not feasible.

There are two main origins of information in identifiers, strings, comments: problem and solution domain knowledge (Marcus et al., 2004). Identifiers or words in strings originating from problem domain knowledge form identification information. By contrast, names originating from solution domain knowledge are *classification information*, i.e. information relevant for classifying a given piece of source code. Ideally, an anonymization approach replaces all identification information while leaving all classification information untouched. This could be achieved by analysis and transformation of the underlying domain model. However, for a legacy system, this is typically not available in any representation (Wagner, 2014).

Therefore, we first use static program analysis to extract the Platform Specific Model (PSM), and a list of all identifiers, which are used as a basis.

Next, we automatically create a replacement mapping for each of the identifiers based on results from the static program analysis. For this, our anonymization algorithm distinguishes three basic types of identifiers: functions, variables and classes. The anonymized identifiers are generated based on the identifier type. For instance, identifiers which represent class names like `BlogProvider` are mapped to `Class_A`, methods like `Blogprovider.Init()` to identifiers like `Class_A.Method_A()`.

Simple relationships like generalization and class-instance can be expressed in the generated identifiers to maintain a certain level of semantics which is typically found in the natural-language relationships of the words used as identifiers. For instance, a class `class Rectangle: Shape` can be represented as `Class_B_extends_Class_A`, an instance variable `Shape* shape = new Shape()` can be represented as `instance_of_Class_A`. Representation of further relationships such as composition or aggregation would require prior creation of a domain model and is therefore not considered in this exploration.

In the pre-processing phase, the source code is prepared for the following renaming phase. Due to the complexity of natural language texts contained in comments, appropriate modifications would require high effort. Thus, comments are stripped from the source code. Like comments, the contents of strings can contain complex natural language texts, in particular product or company names. Therefore, they are replaced by `"String"`. In the renaming phase, remaining strings and identifiers are then replaced according to the previously described mapping.

To assess the readability of the resulting anony-

[4]replacing calls to usually short functions by their body

mized code, we conducted a brief validation experiment. Six employees of an small and medium-sized enterprise software provider (Age min 22, max 50, avg 32.7; Experience min 6, max 29, avg 13.2 years) rated the readability of 10 anonymized source code fragments (length min 7, max 57, avg 27.4 LOC, cf. 7.1) on a five-level Likert scale (measuring agreement between 1 and 5 for: The code is easy to read.) As expected, Obfuscation performed worst (0.7). Our approach (3.7) shows a slight improvement over the naïve approach (3.2).

5 QUALITY CONTROL AND RESULTS AGGREGATION

Crowdsourcing reverse engineering activities produces a set of results from different, potentially unknown contributors. These results may even be contradicting. For companies, the quality of results from CSRE must justify the invested resources. Thus, quality control and results aggregation is crucial.

Considering the classification task described in 2, the amount of correctly classified code fragments should be as high as possible (i.e. high precision and high recall). This depends on several factors. Crowd workers could provide fake answers to maximize their financial reward, leading to poor quality. Different levels of experience among the crowd workers can lead to different classification results on the same code fragment.

The combination of approaches used to achieve good results quality is described according to the schema in (Allahbakhsh et al., 2013) by the following **quality control and results aggregation properties**:

1. Worker selection
2. Effective task preparation
3. Ground truth
4. Majority consensus

Worker selection and effective task preparation are *quality-control design-time approaches*, Ground truth and majority consensus are *quality-control run-time approaches* (Allahbakhsh et al., 2013).

Worker Selection. Since the quality of crowdsourcing results highly depends on the experience of the crowd workers, we use reputation-based worker selection (Allahbakhsh et al., 2013). In most crowdsourcing platforms, crowd workers receive ratings based on the results they contributed to a task. These ratings form the reputation of the crowd worker. Only crowd workers above a specified *reputation threshold* are allowed to complete a task. In our experiments on the bespoke (Mao et al., 2017) crowdsourcing platform microWorkers.com[5], we allowed only workers from the "best workers" group to participate.

Effective Task Preparation. The reverse engineering task has to be described in a clear and unambiguous way and should keep the effort for fake contributions similar to the effort for solving the task. In crowdsourcing research, this is known as *defensive design* (Allahbakhsh et al., 2013). Crowdworkers are provided with a brief description of the classification task and of the available classifications, along with examples. They can refer back to this description at any step of the process.

For the classification, they are presented with the source code and references (as described in 3) with syntax highlighting, the available categories and a text input (cf. Figure 2). In this input, crowd workers have to provide a brief explanation, arguing why they chose a certain classification. Only explanations with more than 60 characters will be accepted. This aims at reducing or at least slowing down fake contributions and allows for filtering during post-processing, e.g. filtering out identically copied explanations. According to our *compensation policy*, crowd workers receive financial and non-financial rewards: After filtering low quality contributions, remaining crowd workers receive a financial reward of 0.30 USD, which corresponds to the platform average during the experiment. Also, they automatically receive a rating of their contribution as non-financial reward, which improves their reputation.

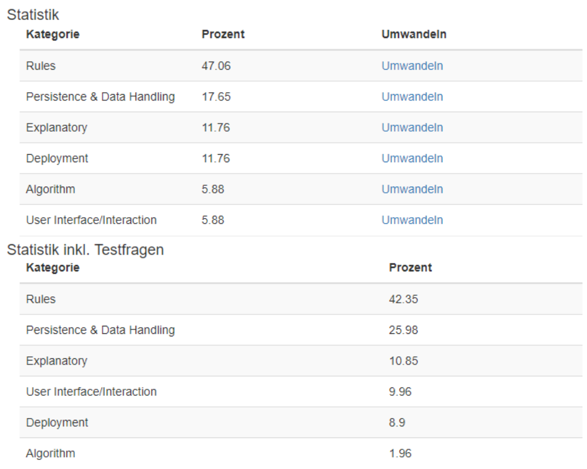
Ground Truth. For the assessment of the quality of a contribution by an individual crowd worker, we employ the ground truth approach: We add previously solved classification tasks with known correct answers into the set of all tasks, which form the ground truth. In this way, assessment of a crowd worker based on the correctness of answers for these test questions can be achieved. We process this information by calculating an *individual user score* $S(w_i) \in [0,1]$ for each crowd worker $w_i \in W$. Comparing the amounts of correct classifications $C^+_{w_i}$ and false classifications $C^-_{w_i}$, the user score is calculated as in 1:

$$S(w_i) = \frac{|C^+_{w_i}|}{|C^+_{w_i}| + |C^-_{w_i}|} \qquad (1)$$

This score can be used as weight factor during results aggregation.

Majority Consensus. To aggregate the crowdsourcing results, we employ the majority consensus technique. For each source code area to be classified, the classifications $C \subset W \times T$ are tuples (w_i, t_k) of a

[5]https://microworkers.com/



Statistik

Kategorie	Prozent	Umwandeln
Rules	47.06	Umwandeln
Persistence & Data Handling	17.65	Umwandeln
Explanatory	11.76	Umwandeln
Deployment	11.76	Umwandeln
Algorithm	5.88	Umwandeln
User Interface/Interaction	5.88	Umwandeln

Statistik inkl. Testfragen

Kategorie	Prozent
Rules	42.35
Persistence & Data Handling	25.98
Explanatory	10.85
User Interface/Interaction	9.96
Deployment	8.9
Algorithm	1.96

Figure 3: Results statistics view.

crowd worker $w_i \in W$ and the type $t_k \in T$ which the crowd worker selected. The resulting voting distribution $V : T \mapsto [0,1]$ is calculated for all possible types $t_i \in T$ as in 2:

$$V(t_i) = \frac{\sum\limits_{(w,t)\in C | t=t_i} S(w)}{\sum\limits_{(w,t)\in C} S(w)} \tag{2}$$

The aggregated result t^* of all crowd classifications is the one with the highest voting value as in 3:

$$t^* = \arg\max_{t\in T} V(t) \tag{3}$$

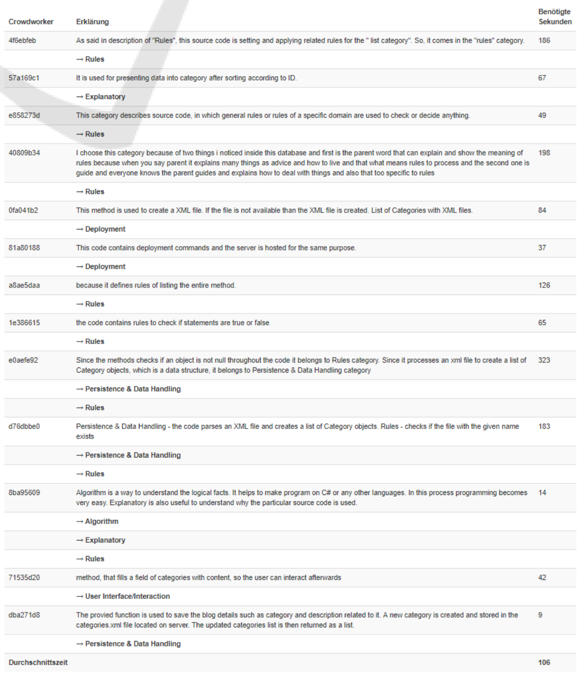
To provide more control, we display an overview with the results distributions and explanations so that it easy to identify and decide edge cases, where no clear majority could be found (cf. figure 3 and 4).

6 RELATED WORK

Research on the application of crowdsourcing for reverse engineering is sparse. Saxe et al. (Saxe et al., 2014) have introduced an approach for malware classification combining NLP with crowdsourcing. While the actual classification work is performed by classical statistical NLP methods such as full-text indexing and Bayesian networks, the initial data is provided by the crowd. The CrowdSource approach creates a statistical model for malware capability detection based on the vast natural language corpus available on question and answer websites like StackExchange. The model seeks to correlate low-level keywords like API symbols or registry keys with high-level malware capabilities like screencapture or network communication. In contrast to our approach, in CrowdSource, crowdsourcing is employed only to generate the required input probabilities for the Bayesian model and not directly for performing the classification work.

Crowdsourcing has seen some consideration in software engineering. For instance, (Nebeling et al., 2012) presents a platform for the crowd-supported creation of composite web applications. Following the mashup paradigm, the web engineer creates the design of the web application based on information and interface components. Nebeling et al. combine a passive and an active crowdsourcing model: *Sharing and Reuse* is realized by providing a community-based component library. It contains public components which can be used by the web engineer to compose the web application. *Active crowdsourcing* is used for the creation of new components. The web engineer defines the required characteristics of the component and makes an open call to a paid, external crowd to provide solution candidates. Improving the technical quality of the crowdsourced contributions is stated as one of the main issues. A good overview on research on crowdsourcing in software engineering can be found in (Mao et al., 2017), which shows a strong increase in this area since 2010.

In the HCI area, crowdsourcing was successfully employed to adapt existing layouts to different screen sizes (Nebeling et al., 2013). The CrowdAdapt approach leverages the crowd for the creation of adapted web layouts and the selection of the best layout variants. Its focus is on end-user development web layout tools driven by the crowd. Crowdsourcing was em-



Crowdworker	Erklärung	Benötigte Sekunden
4f6ebfeb	As said in description of "Rules", this source code is setting and applying related rules for the " list category". So, it comes in the "rules" category.	186
	→ Rules	
57a169c1	It is used for presenting data into category after sorting according to ID.	67
	→ Explanatory	
e858273d	This category describes source code, in which general rules or rules of a specific domain are used to check or decide anything.	49
	→ Rules	
40809b34	I choose this category because of two things i noticed inside this database and first is the parent word that can explain and show the meaning of rules because when you say parent it explains many things as advice and how to live and that what means rules to process and the second one is guide and everyone knows the parent guides and explains how to deal with things and also that too specific to rules	198
	→ Rules	
0fa041b2	This method is used to create a XML file. If the file is not available than the XML file is created. List of Categories with XML files.	84
	→ Deployment	
81a80188	This code contains deployment commands and the server is hosted for the same purpose.	37
	→ Deployment	
a8ae5daa	because it defines rules of listing the entire method.	126
	→ Rules	
1e386615	the code contains rules to check if statements are true or false	65
	→ Rules	
e0aefe92	Since the methods checks if an object is not null throughout the code it belongs to Rules category. Since it processes an xml file to create a list of Category objects, which is a data structure, it belongs to Persistence & Data Handling category	323
	→ Persistence & Data Handling	
	→ Rules	
d76dbbe0	Persistence & Data Handling - the code parses an XML file and creates a list of Category objects. Rules - checks if the file with the given name exists	183
	→ Persistence & Data Handling	
	→ Rules	
8ba95609	Algorithm is a way to understand the logical facts. It helps to make program on C# or any other languages. In this process programming becomes very easy. Explanatory is also useful to understand why the particular source code is used.	14
	→ Algorithm	
	→ Explanatory	
	→ Rules	
71535d20	method, that fills a field of categories with content, so the user can interact afterwards	42
	→ User Interface/Interaction	
dba271d8	The provied function is used to save the blog details such as category and description related to it. A new category is created and stored in the categories.xml file located on server. The updated categories list is then returned as a list.	9
	→ Persistence & Data Handling	
Durchschnittszeit		106

Figure 4: Crowd result details.

ployed primarily as a means of exploring the design space and eliciting design requirements for a multitude of viewing conditions. Unlike other crowdsourcing approaches in software engineering, CrowdAdapt uses unpaid crowd work. Unpaid crowd work can be successfully employed in many HCI contexts due to the high number of users who implicitely contribute feedback through their choices and behavior.

Similar to our approach, CrowdDesign (Weidema et al., 2016) employs the microtasking crowdsourcing model. CrowdDesign uses paid crowd workers from Amazon Mechanical Turk for solving small user interface design problems. The focus is on diversity, i.e. given a set of decision points in the design space, CrowdDesign intends to create various diverse solution alternatives. Early results indicate that good diversity can be easily achieved, but only a small percentage of the crowd-created solutions achieved sufficiently high quality. In contrast, for reverse engineering classification, quality is the most relevant property whereas diversity in the results is not intended.

Larger industrial case studies on the application of crowdsourcing in software development like (Stol and Fitzgerald, 2014) indicate that among the many different activities in software development, those which are less complex and relatively independent are the most successful for crowdsourcing. However, even more complex software development tasks can benefit from the lower costs, faster results creation and higher quality of successful crowdsourcing application. Test automation and modeling of the front end are the two fields in this case study. Similar to our approach, (Stol and Fitzgerald, 2014) focuses on the perspective of an enterprise crowdsourcing customer. A significant number of defects in the produced results indicates quality as one of the main problems. Also, the authors report on problems with continuity since new crowd workers lacked the experience from their predecessors and would at times even re-introduce previously fixed bugs. For these reasons, they conclude that from an enterprise perspective, applicability of crowdsourcing in software engineering is limited to areas which are self-contained without interdependencies, such as GUI design.

Latoza et al. (Latoza and van der Hoek, 2016) identifies eight foundational and orthogonal *dimensions of crowdsourcing for software engineering*: crowd size, task length, expertise demands, locus of control, incentives, task interdependence, task context and replication. Based on these dimensions, they characterize three existing successful crowdsourcing models: peer production, competitions and microtasking. As shown in Figure 5, the Crowdsourcing-based Reverse Engineering Classification described in this paper closely matches the microtasking model. Only two of the eight dimensions are different: while expertise demand in microtasking is generally low, we consider this low to medium for source code classification. Task context, i.e. the amount of information about the entire system required by the worker to contribute, is none for microtasking, compared to low for the classification. This high similarity indicates a high likeliness that microtasking can be similarly successful on the small, independent and easily replicatable source code classification tasks as it already has proven in software testing. Both areas benefit from the high number of workers and the possibility to execute the tasks in parallel. LaToza et al. state that the key benefit of reduced time to market through crowdsourcing can be achieved for models with two characteristics: work must easily be broken down into short tasks and each task must be self-contained with minimal coordination demands. Our approach meets both of these characteristics.

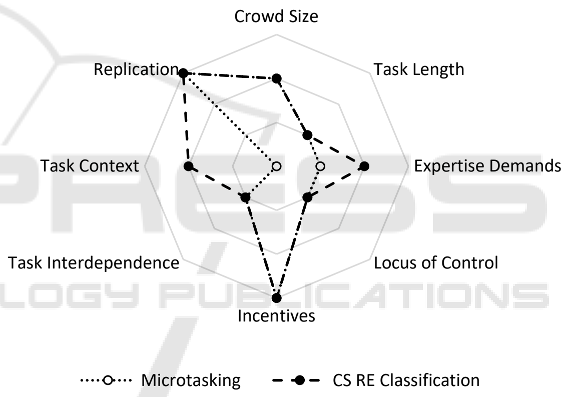


Figure 5: Comparing Microtasking and Crowdsourced Reverse Engineering (CS RE) Classification.

Satzger et al. (Satzger et al., 2014) describe a distributed software development which abstracts the workforce as crowd. The private crowd consists of company employees whereas the public crowd is provided by crowdsourcing platforms. Aiming at collaborative crowdsourcing for the creation of software in enterprise contexts, the proposed approach starts with requirement descriptions in customer language which are than transformed into developer tasks for the crowd by a software architect. These tasks are then delegated to private and public crowds, developed collaboratively. Crowd workers can furthermore recursively divide tasks into smaller tasks and delegate these tasks to the crowd. The development process is iterative and tries to combine properties and artifacts of agile development methodologies with collaborative crowdsourcing. Similarly, our approach integrates with agile development, however, due to the nature of the reverse engineering classification task, it is not collaborative.

7 EVALUATION

In this section we report on our experiences from an evaluation experiment which was conducted on crowdsourcing platform microWorkers.

7.1 Experimental Design

To evaluate our proposed approach, we automatically extracted and randomly selected 10 source code fragments from the open source project BlogEngine.NET[6], an ASP.NET based blogging platform published under the Microsoft Reciprocal License (MS-RL)[7]. The 10 code fragments had a length between 7 and 57 LOC, on average 25.4 LOC. We manually classified each of them using the following 8 categories from our source code knowledge ontology (Heil and Gaedke, 2016):

1. Business Process
2. Algorithm
3. Persistence & Data Handling
4. User Interface & Interaction
5. Explanatory
6. Rule
7. Configuration
8. Deployment

These categories extend the 3 basic categories typically considered (presentation, application logic and persistence (Canfora et al., 2000)) allowing a more detailed distinction of the knowledge contained in source code. We implemented our approach in a prototype by extending our existing source code annotation platform (Heil and Gaedke, 2016). Crowd worker views and authentication mechanisms, classification task extraction based on doxygen[3] and integration with crowdsourcing platform microWorkers[5] were implemented.

The crowd worker view was tracking the time which the crowd worker spent on it, using `focus` and `blur` events. We started a classification campaign which ran for 14 days. Only workers from the "best workers" group were allowed to participate. The financial reward of 0.30 USD was paid for each set of 3 classifications.

[6]http://www.dotnetblogengine.net/

[7]https://opensource.org/licenses/MS-RL

7.2 Results

During the experiment, 34 unique crowd workers contributed 187 classifications on our test data set. Table 1 shows the results. F are the ten code fragments, Categories 1 to 8 correspond to the 8 categories introduced before, $|C|$ is the number of classifications and $|W|$ the number of crowd workers. The numbers in the categories cells represent the number of classifications which classified this code fragment as belonging to this category. Highlighted in bold are the maximum values, which are the basis for the majority consensus. Highlighted with grey background is the correct classification of the code fragment. Note that the number of crowd workers and classifications can be different, because we allowed to assign more than one category per fragment. The length of the code fragment in LOC is indicated by l, Σt represents the overall time in seconds which crowd workers spent on the crowd view of the code fragment, $\bar{t}$ is the average time. The error rate f_e (cf. 4)

$$f_e = \frac{|C^-|}{|C|} \tag{4}$$

is the ratio of false classifications to all classifications of a code fragment.

To investigate the degree of agreement or disagreement between the crowd workers classifications, we include the entropy E (cf. 5)

$$E = -\sum_{i=1}^{k} f_i \lg f_i \tag{5}$$

and the normalized Herfindahl dispersion measure H^* (cf. 6)

$$H^* = \frac{k}{k-1}\left(1 - \sum_{i=1}^{k} f_i^2\right) \tag{6}$$

based on the relative frequencies f_i of the classifications in the $k = 8$ classes. Entropy and Herfindahl measure represent the disorder or dispersion among crowd workers' classifications, a unanimous classification result yields $E = 0$ and $H^* = 0$. The higher the disagreement between the crowd workers, the more different classifications, the closer E and H^* get to 1. Therefore, they can be seen as indicators of the certainty of the classification across the crowd workers. On average, 16 crowd workers created 18.7 classifications per code fragment.

7.3 Discussion

The average error rate was 0.655, which seems high at first glance. However, due to the majority consensus method, 7 of 10 code fragments could be correctly classified. The minimum error rate was 0.25

Table 1: Experimental Results and descriptive statistics.

	Categories																
F	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	$\|C\|$	$\|W\|$	l	Σt	$\bar{t}$	f_e	E	H^*
A	1	**14**	4	1	1	1	1	1	24	19	18	2822	122	0.4167	0.6113	0.6906
B	0	1	3	0	0	**12**	0	0	16	16	20	2531	158	0.25	0.3053	0.4427
C	3	0	**4**	1	0	1	0	0	10	10	40	1128	112	0.6	0.6160	0.8
D	2	5	**6**	3	0	5	2	0	23	21	8	3033	131	0.7391	0.7402	0.8948
E	4	2	1	**9**	2	0	3	1	22	18	7	2580	117	0.5909	0.7228	0.8448
F	0	0	3	0	1	6	**7**	2	19	15	28	2857	150	0.6316	0.6146	0.8064
G	3	1	1	2	2	**6**	0	0	15	13	57	3225	215	0.8667	0.6891	0.8395
H	0	2	**12**	2	4	3	2	2	25	21	24	5249	209	0.52	0.6541	0.7893
I	0	1	3	1	2	**8**	0	2	17	13	40	1917	112	1	0.6504	0.7920
J	2	2	**4**	0	1	2	1	**4**	16	14	12	3393	212	0.9375	0.7902	0.9115

on fragment B and the maximum 1 for fragment I. Assuming little variation in the expertise of the participating crowd workers, this points to differences in the classification difficulty (fragment I was one of the longest) and in the understanding of the categories. Regarding the categories, Rule was the most frequent classification with 23.5%, followed by Persistence & Data Handling (21.9%), Deployment was the least frequent category (5.3%). Business Process and Explanatory did not get majorities in the fragments where they were the correct result, indicating that they might not be clear enough for the crowd workers. All other categories were correctly classified by the respective majorities.

Average entropy was calculated at 0.639 and average Herfindahl dispersion measure at 0.757. The minima of both co-occur with the minimal error rate, their maxima with the second-highest error rate. We found a significant ($\alpha = 0.05$) positive correlation (Pearson's $\rho = 0.724$, $p = 0.018$) between the error rate and the entropy and between error rate and Herfindahl dispersion measure ($\rho = 0.757$, $p = 0.011$). Possible interpretation: the more crowd workers chose one classification, the less likely it is a wrong classification. Clear majorities for wrong classifications were not observed in our experiment. This underlines the basic *"wisdom of the masses"* principle of crowdsourcing in gerneral and the assumption of majority consensus, that majorities are indicative of correct answers.

Our experiment did not show a correlation between the length of a code fragment and the time the crowd workers needed for classification. This indicates influence of another variable and can be interpreted by assuming different levels of difficulty/clarity of the classification of the code fragments.

Fragment J was classified as 3 (Persistence) or 8 (Deployment) by the majority. In the texts from the explanation field, crowd workers argued that it is related to persistence because the class from which the fragment was extracted is related to persistence (XML or DB based). This was a very interesting observation to us, because our dataset did not inlcude the entire class source code. Thus, several crowd workers have looked up the sample source code on the internet and read also the surrounding parts in order to classify. We were positively surprised by this level of active engagement and investment in time by the crowd workers in order to complete their task.

Our experiment has shown that the the expertise level of the best crowd workers group on crowdsourcing platform microWorkers in combination with our quality control is sufficient to perform the reverse engineering classification activity and produce decent results. The overall degree of correctness of 70% is a good result similar to what can be achieved by a single expert performing the same task. However, with less than 20 USD expenses for classifying the ten code fragments, crowdsourcing is a significantly more cost-effective solution. The results indicate that crowdsourcing can be applied to perform specific reverse engineering activities, when they are broken down into small tasks and the process is guided by suitable quality control methods. Larger-scale experimentation could look deeper into the applicability of measures for disagreement as indicators for correctness, into suitability of other crowds from different platforms and into understanding the complexity of different reverse engineering tasks for crowd workers.

8 CONCLUSION

In this paper, we introduced the idea of crowdsourced reverse engineering and identified three major challenges – 1) automatic task extraction, 2) source code anonymization and 3) quality control and results aggregation – for applying crowdsourcing in the reverse engineering domain. We illustrated these challenges in relation to the reverse engineering problem of concept assignment by presenting our approach based on crowdsourced classifications. For each of the challenges, we presented the main properties of suitable methods and described how we addressed these in our approach. In particular, we have shown that balancing readability and anonymization requirements in source code anonymization is challenging and that it is an interesting field for further research to find more sophisticated methods. We demonstrated a suitable classification task extraction method re-using existing software documentation tools. Regarding the quality assurance and aggregation of the crowdsourced results, we showcased a method based on a combination of several crowdsourcing quality control methods.

In the overview on existing literature, we identified a lack of consideration of crowdsourcing for reverse engineering, but also demonstrated the similarity of crowdsourced concept assignment to microtasking in eight dimensions and provided examples of successful application of crowdsourcing in software engineering. This matching procedure can be used as a blueprint for identifying further reverse engineering activities and corresponding crowdsourcing paradigms to explore their crowdsourced realization in future work.

We reported on our experiences from an evaluation experiment on the microWorkers crowdsourcing platform, which produced 187 results by 34 crowd workers, classifying 10 code fragments at a low cost. The quality of the results indicates that crowdsourcing is a suitable approach for certain reverse engineering activities. We were positively surprised by some observations which showed an unexpectedly high level of engagement and effort by individual crowd workers to provide good solutions. By calucation of entropy and Herfindahl dispersion measure, we could see some evidence for the applicability of the wisdom of the masses crowdsourcing principle in our context, as higher levels of agreement across the crowd workers was indicative of correctness.

The next challenge is to see, how similar results can be achieved in other areas of reverse engineering or the quality of the results in the described approach can be further improved. A larger scale evaluation should yield more insights into the applicability of crowdsourcing for reverse engineering activities, in particular when combined with more specific, tailored measures of agreement in crowdworker results. One very interesting field is the specification of concrete problem and solution domain models by the crowd. It has to be investigated if this is possible through isolated microtasking using a more comprehensive classification ontology specific to the legacy system instance, or whether complex collaborative crowdsourcing approaches are required. While anonymization has been demonstrated as the most difficult challenge providing many opportunities for further research, investigation of the application of our proposed method in contexts without anonymization requirements such as intra-organzation settings or open source projects can produce further insights.

ACKNOWLEDGMENTS

This research was supported by the eHealth Research Laboratory funded by medatixx GmbH & Co. KG.

REFERENCES

Allahbakhsh, M., Benatallah, B., Ignjatovic, A., Motahari-Nezhad, H. R., Bertino, E., and Dustdar, S. (2013). Quality control in crowdsourcing systems: Issues and directions. 17(2):76–81.

Arboit, G. (2002). A method for watermarking java programs via opaque predicates. In *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*, pages 102–110.

Aversano, L., Canfora, G., Cimitile, A., and De Lucia, A. (2001). Migrating legacy systems to the Web: an experience report. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 148–157. IEEE Comput. Soc.

Biggerstaff, T., Mitbander, B., and Webster, D. (1994). The concept assignment problem in program understanding. In *Proceedings of 1993 15th International Conference on Software Engineering*, volume 37, pages 482–498. IEEE Comput. Soc. Press.

Canfora, G., Cimitile, A., De Lucia, A., and Di Lucca, G. a. (2000). Decomposing legacy programs: a first step towards migrating to clientserver platforms. *Journal of Systems and Software*, 54(2):99–110.

Ceccato, M., Di, M., Falcarin, P., Ricca, F., Torchiano, M., and Tonella, P. (2014). A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, 19(4):1040–1074.

Heil, S. and Gaedke, M. (2016). AWSM - Agile Web Migration for SMEs. In *Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering*, pages 189–194. SCITEPRESS - Science and and Technology Publications.

Heil, S. and Gaedke, M. (2017). Web Migration - A Survey Considering the SME Perspective. In *Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 255–262. SCITEPRESS - Science and Technology Publications.

Kazman, R., Brien, L. O., and Verhoef, C. (2003). Architecture Reconstruction Guidelines, Third Edition.

Technical Report November, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Latoza, T. T. D. and van der Hoek, A. (2016). Crowdsourcing in Software Engineering : Models , Opportunities , and Challenges. *IEEE Software*, pages 1–13.

Mao, K., Capra, L., Harman, M., and Jia, Y. (2017). A survey of the use of crowdsourcing in software engineering. *Journal of Systems and Software*, 126:57–84.

Marcus, A., Sergeyev, A., Rajlieh, V., and Maletic, J. I. (2004). An information retrieval approach to concept location in source code. *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 214–223.

Nebeling, M., Leone, S., and Norrie, M. (2012). Crowdsourced Web Engineering and Design. In *Proceedings of the 12th International Conference on Web Engineering*, pages 1–15, Berlin, Germany.

Nebeling, M., Speicher, M., and Norrie, M. (2013). CrowdAdapt: enabling crowdsourced web page adaptation for individual viewing conditions and preferences. *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing system*, pages 23–32.

Rose, J., Jones, M., and Furneaux, B. (2016). An integrated model of innovation drivers for smaller software firms. *Information & Management*, 53(3):307–323.

Satzger, B., Zabolotnyi, R., Dustdar, S., Wild, S., Gaedke, M., Göbel, S., and Nestler, T. (2014). Chapter 8 - Toward Collaborative Software Engineering Leveraging the Crowd. In *Economics-Driven Software Architecture*, pages 159–182.

Saxe, J., Turner, R., and Blokhin, K. (2014). CrowdSource: Automated inference of high level malware functionality from low-level symbols using a crowd trained machine learning model. In *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, pages 68–75. IEEE.

Stol, K.-J. and Fitzgerald, B. (2014). Two's company, three's a crowd: a case study of crowdsourcing software development. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 187–198, New York, New York, USA. ACM Press.

Wagner, C. (2014). *Model-Driven Software Migration: A Methodology*. Springer Vieweg, Wiesbaden.

Warren, I. (2012). *The renaissance of legacy systems: method support for software-system evolution*. Springer Science & Business Media.

Weidema, E. R. Q., López, C., Nayebaziz, S., Spanghero, F., and van der Hoek, A. (2016). Toward microtask crowdsourcing software design work. In *Proceedings of the 3rd International Workshop on CrowdSourcing in Software Engineering - CSI-SE '16*, pages 41–44, New York, New York, USA. ACM Press.