Crowdsourced Reverse Engineering: Experiences in Applying Crowdsourcing to Concept Assignment

Sebastian Heil(✉), Valentin Siegert, and Martin Gaedke

Technische Universität Chemnitz, 09107 Chemnitz, Germany
{sebastian.heil,valentin.siegert,martin.gaedke}@informatik.tu-chemnitz.de

Abstract. This article details the idea of Crowdsourced Reverse Engineering (CSRE) by analysing three major challenges: (1) automatic task extraction, (2) source code anonymization and (3) results aggregation and quality control. We re-formulate the Reverse Engineering activity of concept assignment as a crowdsourced classification task to exemplify these challenges and describe suitable methods to address them. Our overview on existing research of crowdsourcing showcases examples of successful application in the field of Software Engineering and argues that Reverse Engineering activities like Concept Assignment are likely to also benefit from crowdsourcing by determining a high similarity in eight crowdsourcing dimensions to the microtasking model. Our experiments on the crowdsourcing platform microworkers.com support this, producing 187 results by 34 crowd workers which classified 10 code fragments with decent quality. We provide an extended analysis of the observed crowd workers' behavior and report evidence of surprisingly high levels of engagement and efforts undertaken by the crowd. Concluding our experiences, this article indicates three open research challenges for future work.

Keywords: Reverse engineering · Crowdsourcing · Microtasking · Concept assignment · Classification · Web migration · Software Migration

1 Introduction

Software Migration to the Web is a crucial challenge for software developing companies with legacy systems. Changing expectations of users towards modern software pose new challenges for existing software systems that are not web-based. These challenges are particularly rooted in the diversity of user interactions of recent web applications. The continuous evolution of web technologies and the termination of support for obsolete technologies furthermore increase the pressure to modernize non-web legacy systems [7,26]. With web browsers becoming the standard interface for many applications, web applications provide a solution to platform-dependence and deployment problems [4]. Software developing companies are aware of these benefits and reasons for web migration.

On the other hand, in particular Small and Medium-sized Enterprises (SMEs) face difficulties when trying to commence a web migration [12].

Our problem analysis based on LFA[1] problem trees identified a variety of sub-problems which can be summarized under two main factors: *doubts about feasibility* and *doubts about desirability* [13], which render SME-sized software developing companies hesitant to migrate their existing software products to the web. While we addressed doubts about desirability in [13], this work focuses on doubts about feasibility, which are mainly originating in the danger of losing knowledge throughout the migration process [7].

Small and medium-sized software providers often tailor their successful software products specifically to a certain niche domain, resulting from years of requirements engineering [21]. Therefore, the amount of valuable domain knowledge from the problem and solution domain [18] such as models, processes, rules, algorithms etc. encoded in the source code is vast. [26] Due to the paradigm shifts introduced through webmigration – client-server separation in the spatial and technological dimension [7], asynchronous request-response-based communication [4], explicitly addressable application states via URLs and navigation to name but a few – redevelopment methods bears the risk of losing this knowledge.

The problem and solution domain knowledge in legacy systems, however, is only implicitly represented by the source code and often poorly documented [26,27]. Therefore, *Reverse Engineering* is required to elicit this knowledge, to make it explicit and thus available for subsequent web migration processes. For small and medium-sized enterprises, existing re-documentation approaches [14] are not feasible since they cannot be integrated into day-to-day agile development activities and require additional human resources. Therefore, we introduced an approach based on in-situ source code annotations [11], allowing to enrich the legacy source code by directly linking parts of code with explicit representations of the knowledge which is contained in them. Web engineers are enabled to reference the elicited knowledge in emails, wikis, task descriptions etc. and to jump directly to their definition and location in the legacy source code, using a web-based annotation platform. The identification of domain knowledge in source code is known as *Concept Assignment* [6].

Through integration into the daily development activities of the small and medium-sized enterprise, concept assignment supported by our platform allows to incrementally re-discover and document the valuable domain knowledge. However, the concept assignment activity itself is manual. This requires a high amount of effort and time, in particular taking into account the limited resources of small and medium-sized enterprises. Moreover, the results of manual concept assignment depend solely on the migration engineer executing the activity.

The concept assignment process involves reading a part of legacy source code, selecting a relevant portion called region of interest (ROI) and determining the type of knowledge which this ROI represents, before further manual or automatic code analysis can be applied to extract model representations of the knowledge. This process can be considered a *classification task*. Crowdsourcing has been successfully applied to solve various classification tasks in areas like

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

image or natural language text classification. Also in the context of software engineering, crowdsourcing methods have been reported successful, in particular on smaller tasks without interdependencies [17,25], like the above. Thus, in this article we explore *Crowdsourced Reverse Engineering* (CSRE) through experimentation with the reverse engineering activity of identifying different types of knowledge in legacy codebases. This article is an extended and revised version of our previous work in [10]. Our experiments identified three main challenges. This article details these challenges, reports on how we addressed them and provides the results of our evaluation. Since these challenges are not specific to concept assignment, which is used as an example for this work and can be encountered when applying crowdsourcing to other reverse engineering activities.

Challenges of the Application of Crowdsourcing in Reverse Engineering:

1. Automatic Extraction and Creation of Crowdsourcing Tasks from the Legacy Source
2. Balancing Controlled Disclosure of Proprietary Source Code with Readability
3. Aggregation of Results and Quality Control

Existing crowdsourcing platforms require suitable classification tasks. These can be derived by splitting the legacy source code into fragments which can then be classified by the crowd workers. The fragment size has to balance context with classification, i.e. they should be large enough to provide sufficient context for a meaningful classification and small enough to allow for a unambiguous classification and a good overall recall in relation to the entire code base. Moreover, the legacy source is a valuable asset of the company. Thus, disclosure of code fragments on a public crowdsourcing platform to a potentially unknown audience needs to be controlled.

Competitors should be prevented from identifying the authoring company, the concrete software product or even the application domain, in order to not allow them to gain insights on the software product or – as worst case – replicating parts of it. On the other hand, a suitable anonymization method needs to be balanced with the source code readability. Code which is produced by traditional code obfuscation algorithms is intendedly hard to read [8]. This would jeopardize achieving high quality classification results by the crowd workers. Aggregating the crowd workers' classification results with effective quality control measures is therefore a key challenge. Fake contributions need to be filtered and contradicting classifications have to be resolved. While manual quality control by the crowdsourcing company would be effective, the advantage gained by crowdsourcing would be mitigated by the high manual effort to ensure a decent classification quality.

This article reports on our experiences when applying crowdsourcing in the reverse engineering domain. We outline the CSRE approach in Sect. 2 and detail the three challenges of automatic task extraction in Sect. 3, source code anonymization in Sect. 4 and quality control and results aggregation in Sect. 5. In Sect. 6, we position CSRE against existing work, report on and analyse the results from our experiments in Sect. 7 and conclude the article with an outlook on open issues in Sect. 8.

2 The CSRE Approach

To demonstrate the application of crowdsourcing in the domain of reverse engineering, we experimented with the reverse engineering activity of *Concept Assignment*. The original problem of concept assignment has been defined by Biggerstaff et al. in [5]. It aims at reconstructing "human-oriented expressions of computational intent" by identifying the concepts and assigning them to "the specific implementation structures within the program" [5]. The concept assignment problem is "the problem of discovering these human-oriented concepts and assigning them to their realizations within a specific program or its context" [5].

Concept assignment research has investigated a variety of different concepts: from concrete domain concepts [6] to features [18] to abstract concerns [9]. An important distinction is between *problem domain concepts* and *solution domain concepts*: while problem domain concepts (e.g. processes, rules, business entities) originate in the domain for which the software was built, solution domain concepts (e.g. algorithms, patterns) are from the domain of programming [18].

In Fig. 1, we reformulated the concept assignment process [6] as a classification problem: first the source code is read (for manual concept assignment as in [9]) or parsed (for automatic methods like [18]), regions of interest are identified and the represented concept determined. Optionally, this is followed by manual or automatic extraction of a formal representation of the concept (e.g. in UML, BPMN) to allow subsequent use of model-driven methodologies.

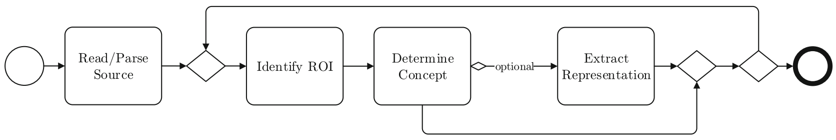


Fig. 1. Concept assignment process reformulated as classification.

Manual concept assignment often is a lexical search-based activity [24] using identifiers [18] and can be supported by tools like ConcernTagger [9] or Annotation Platform [11]. Recent automatic concept assignment approaches are Information Retrieval (IR) techniques employing Natural Language Processing (NLP) methods [24]. The filtering technique in [1] applies NLP analysis to identifiers of classes, methods and attributes to extract domain ontologies, [18] presents an approach based on Latent Semantic Indexing, [24] uses action-oriented identifier graphs. The possibility of applying crowdsourcing to concept assignment has not been considered yet. Thus, we use concept assignment as basis for our experimental application of crowdsourcing in reverse engineering.

The crowdsourcing-based classification process for concept assignment in legacy code bases presented in Fig. 2 involves three roles:

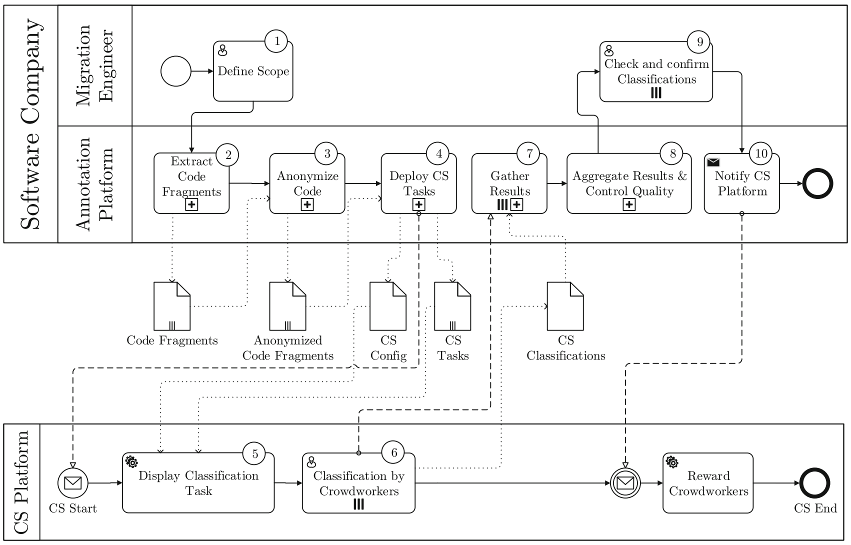
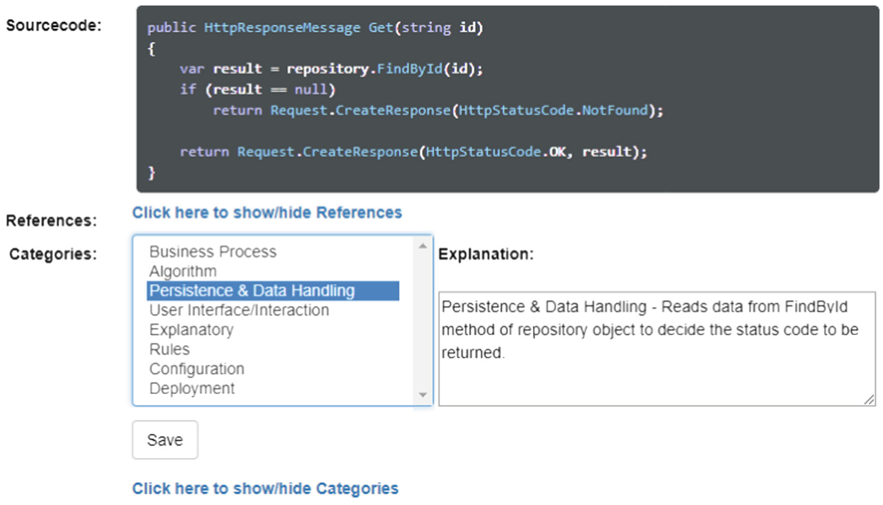


Fig. 2. Crowdsourcing-based source code classification process.

1. *Migration Engineer* is the person conducting the web migration
2. *Annotation Platform* is a system role that represents a web migration support platform [11] for the concept assignment
3. *CS Platform* represents a suitable crowdsourcing marketplace, allowing to post an open classification call to the crowd

The process starts by the migration engineer (1) defining the scope of code to be classified on the legacy code base. This scope can be defined in terms of a subset of concrete source files, software components (project/solution files) or the entire code base. The annotation platform (2) automatically extracts code fragments for classification as described in Sect. 3. The extracted fragments are (3) pre-processed to achieve the intended anonymization properties, which we describe in Sect. 4. For each of the anonymized code fragments, the annotation platform (4) deploys classification tasks in the CS Platform. The CS Configuration data passed to the CS Platform to set-up the tasks includes a brief description of the concept assignment classification task, a URL pointing to the classification view for crowdworkers (Fig. 3) in the annotation platform, the requirements for selecting suitable crowd workers and the reward configuration. Matching crowdworkers according to the crowdworker requirements are (5) presented a textual description of the available categories for classification, following our ontology for knowledge in source code [11]. Depending on the terms of use of the CS platform and its technological capabilities, the URL of the crowd worker view is either presented as a link or loaded in the CS platform using an iframe.

The crowd worker view displays the code fragment to be classified and the selection control of possible categories to the crowd worker in order to support and capture his classification result (6). In addition, it shows a list of source


Sourcecode:
References:
Click here to show/hide References
Categories:
Business Process
Algorithm
Persistence & Data Handling
User Interface/Interaction
Explanatory
Rules
Configuration
Deployment
Explanation:
Save
Click here to show/hide Categories

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

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

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

Fig. 3. Crowd worker view [10].

code references to allowing the crowdworker to review those parts of source code which are referenced in the code fragment. (7) User-specific URLs and temporary tokens are employed to capture the results per crowdworker and authenticate access to the annotation platform. In (8), the classification results are aggregated across the different crowdworkers and quality control measures are applied according to Sect. 5. The filtered results can then be (9) automatically included in the annotation platform, or marked for further review by the migration engineer, allowing to accept or reject them. Finally, the annotation platform (10) notifies the CS platform to reward the participating crowd workers according to the reward policy. The following three sections provide details about CSRE's components addressing the main challenges raised in the introduction.

3 Classification Task Extraction

3.1 Problem Analysis

According to [25], typical tasks for crowdsourcing, called *Micro-tasks*, are characterized as self-contained, simple, repetitive, short, requiring little time, cognitive effort and specialized skills. Of these characteristics, classification of code fragments as described in Sect. 2 matches the first five. Classification results are not dependent on other classification results. The classification act itself is a simple selection from a list of available options. The classification activity is highly repetitive and a single classification can be achieved in relatively low time. The last two characteristics are slightly different: The cognitive effort required is higher compared to other successfully crowdsourced classification tasks like image classification. To some extent, specialized skills are required since the crowd workers have to have a sufficient capability to read and understand source code in the legacy code base's programming language. However, only a basic understanding of programming and limited knowledge of the programming language are sufficient to read and understand enough of a given code fragment to determine the correct classification. Thus, the skill requirements are not extremly high and the concept assignment classification activity is suitable for a much wider range of crowd workers compared to crowdsourcing the entire development of an application as in [22].

To automatically extract micro classification tasks from the legacy code base, it has to be divided into code fragments for classification. These are identified through structural code analysis. Suitable methods serving this purpose must have three essential **classification task extraction properties**:

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

Automation. To perform the analysis and to carry out the identification of relevant code fragments for classification, no additional user interaction should be required by a suitable extraction method.

Legacy Language Support. Since structural code analysis is specific to the programming language, the method should support common programming languages. According to IEEE Spectrum[2], the ten most widely used programming languages are: Python, C++, C, Java, C#, PHP, R, JavaScript, Go and Assembly. While Go is a relatively new language (appeared in 2009) and Javascript has only recently seen an increased use in the context of web applications, R is a language mainly used for statistics and data analysis. These languages are therefore not considered relevant for a web migration. Typical languages found in legacy software to a larger extent include C, C++, Java and Assembly and should therefore be supported.

Completeness of References. To provide a crowd worker with sufficient information to properly categorize a code fragment, the control and data flow must be understandable from the code provided. To display this information to the crowd worker, the extraction method must include information about source code that is referenced in the code fragment.

3.2 Solution

We analyzed three groups of approaches for classification task extraction. *Documentation tools* are originally used to automatically create source code documentation. Existing documentation tools can be re-used for task extraction, however, instead of developing specific extraction tools. To create the documentation, the

[2] https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages.

structure of the source code is analyzed and transformed into representation formats. Thus, this group of methods allows to identify structural properties of a source code. *Syntactic analysis tools* explicitly analyze code regarding its structural properties. Two different types exist: regular-expression-based and parsers. Regular expressions are used to search for patterns in texts. Thus, a suitable set of regular expressions allows identifying relevant code fragments for classification. Parsers create representations of the syntactical structure of a program, based on abstract syntax trees, describing the structure and sequence of statements of the program code. Similar to documentation tools, *Syntax highlighting tools* generate custom representations of the structure of source code in order display syntax highlighting in editors and IDEs.

The applicability of these three groups for the extraction of classification tasks was systematically investigated by evaluating them against the three aforementioned essential properties as requirements. Production-grade implementations of documentation tools exist for most programming languages. Referenced parts of source code are completely traceable, ensuring good understanding for crowd workers. Through the capability to configure the extraction process by command line parameters, automation is easily achieved. Being a standardized means of extracting information from text, regular expressions are supported by all current programming languages and can thus be employed for automatic extraction, as part of a tailored extraction program. However, tracing source code references can would require very high effort and complex iterative use regular expressions. On the other hand, parsers allow to track source code references by analyzing data and control flow and are available for most programming languages. The analysis results generated in the parsing process, however, are either not exportable or they are available as graphical representations only. This makes further processing difficult. Therefore, using parsers in an automated extraction process is significantly limited. While syntax highlighting tools allow the identification of code fragments and internally create a representation of code structure, exporting these structure file to elicit source code references is restricted to certain platforms. As a result, their applicability is limited.

Based on the above considerations and supported by an internal feasibility study by students, we decided to use documentation tools as basis for the fully automated classification task extraction. The implementation runs "Doxygen"[3] on the legacy code base and parses the generated documentation to identify relevant code fragments and referenced source code.

4 Source Code Anonymization

4.1 Problem Analysis

The crowdsourcing paradigm implies that work is not assigned to individual workers, but instead crowd workers respond to an open call. The group of workers is potentially large and they are unknown to the company. [15] Thus, posting a

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

task on a crowdsourcing platform is equivalent to publishing the task contents, bearing the risk that competitors access the code fragments from the crowd tasks and use them uncontrolledly.

Proper *source code anonymization* means are therefore relevant to allow companies to successfully employ CSRE. *Code obfuscation* techniques adapt an existing source code to make it harder to understand/reverse engineer, while maintaining the original functionality [8]. Thus, they would provide a partial solution to prevent unintended distribution of a company's valuable source code. However, the readability is also severely impacted. [8] assesses the impact of code obfuscation techniques on the understandability through human readers.

In the context of CSRE, the challenge of source code anonymization is to balance information disclosure with readability: While a suitable anonymization method sufficiently modifies the code fragments to prevent unintended use, it has to maintain readability to a level that allows crowd workers to achieve sufficient understanding of the code to perform the reverse engineering task.

The following **anonymization properties** reflect this necessary balance. A suitable anonymization method must:

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

We analyze these three anonymization properties in the following. This section is not structured per property because achievement of any of the properties influences the others. Obfuscation techniques employing code optimizations like inline expansion[4] or adding artificial branches to the control flow (cf. opaque predicates [3]) alter the syntactic sequence of expressions. As the control flow is not maintained, these obfuscation techniques are disregarded.

Identifier renaming has shown good results in source code obfuscation [8]. Identifiers are, however, not the only constituents of code containing information relevant for the identification of software provider, software product or application domain (*identification information* in the following). The three different *loci of identification information* are: identifiers, strings and comments. While traditional code obfuscation has to produce identical software from end users' perspective, anonymization for classification can apply modifications to string contents: The anonymized source code is only displayed to crowd workers for reverse engineering, but not used to compile into software and used by end users.

Another difference to traditional identifier renaming, which typically yields intendedly meaningless random combinations of characters and numbers, is that source code anynomization replacements in the context of CSRE have to sufficiently maintain readability and information content. The naïve approach would be dictionary-based replacements: creating custom blacklists of words and defining mappings to their respective replacements. However, completeness of the

[4] Replacing calls to usually short functions by their body.

anonymization would highly depend on the completeness of theses dictionaries and the approach requires high manual effort. This is not feasible for larger code bases as found in the professional software production context of small and medium-sized enterprises.

According to the two main origins of information in identifiers, strings and comments, research distinguishes problem and solution domain knowledge [18]. Identification information is found in identifiers or words in strings originating from the problem domain. Solution domain knowledge represents *classification information*, i.e. information relevant for classifying a code fragment. An ideal anonymization approach replaces all identification information while maintaining all classification information. Transformation of the domain model would allow this. For a legacy system, however, this model is typically not available [26].

4.2 Solution

We use static program analysis to extract the Platform Specific Model (PSM), and a list of all identifiers. Based on results from the static program analysis, we automatically generate a replacement mapping for each of the identifiers. Our anonymization algorithm (cf. Algorithm 1) generates these replacement mappings based on the identifier type. It distinguishes three basic types of identifiers: functions, variables and classes. For example, identifiers representing class names like `BlogProvider` are mapped to `Class_A`, methods like `Blogprovider.Init()` to identifiers like `Class_A.Method_A()`. Relationships between concepts are understandable to human readers through relationships in natural language between corresponding identifiers. Through identifier renaming, these relationships get lost. To improve understanding, simple relationships like generalization and class-instance can be expressed in the generated identifiers to maintain a certain level of semantics. For example, a class `class Rectangle: Shape` can be replaced as `Class_B_extends_Class_A`, an instance variable `Shape* shape = new Shape()` can be renamed to `instance_of_Class_A`. Representing further relationships like composition or aggregation would require an existing domain model. Prior creation of a domain model contradicts the aim of concept assignment, therefore these are not considered.

The pre-processing phase prepares the source code for the following renaming phase. Due to the complexity of Natural language texts contained in comments can be complex. Appropriate modifications would require high effort, thus, comments are stripped from the source code. Likewise, strings contents can contain complex natural language texts, potentially containing product or company names. Therefore, they are replaced by `"String"`. In the renaming phase, remaining strings and identifiers are replaced according to the mapping described above.

Assessing the readability of the resulting anonymized code, we conducted a brief validation experiment. Six employees of an SME-sized 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. Sect. 7.1) on a five-level Likert scale (measuring agreement between 1 and 5 to “The code is easy to read”). Expectedly, traditional obfuscation was rated near-unreadable (0.7) whereas CSRE (3.7) performed slightly better than the naïve approach (3.2).

Algorithm 1. CSRE Anonymization Algorithm.

```
    Input: Source Code $S$, Platform Specific Model $PSM$, Identifier List $I$
    Output: Anonymized Source Code
1   $m(i) = \begin{cases} \text{“instance\_of\_”} + m(c) \textbf{ if } i \text{ instance of } c \\ \text{genericName}(i) + \text{“\_extends\_”} + m(s) \textbf{ if } i \text{ subclass of } s \\ \text{genericName}(i) \textbf{ else} \end{cases}$
2   replace Strings in $S$ by “String”
3   remove comments from $S$
4   replace all identifiers $i \in I$ in $S$ with $m(i)$
5   return $S$
```

5 Results Aggregation and Quality Control

5.1 Problem Analysis

Crowdsourcing produces a set of results from different and unknown contributors. These results may even be contradicting. The quality of results from CSRE must, however, justify the resources invested by the company. Ensuring quality of crowdsourced results is a challenge. [19,25,28] Thus, proper results aggregation and quality control is crucial.

For the classification task described in Fig. 1, the amount of correctly classified code fragments should be as high as possible, i.e. good precision is required. The precision depends on several factors: Crowd workers sometimes provide fake answers to minimize their effort, leading to poor quality. Different experience levels of the crowd workers can lead to different classification results on the same code fragment.

To aggregate results and achieve good quality, several *quality-control design-time approaches* (Worker selection, Effective task preparation) and *quality-control run-time approaches* (Ground truth, Majority consensus) (cf. [2]) are considered.

Quality Control and Results Aggregation Properties:

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

5.2 Solution

In the following, we outline the combination of approaches used to achieve good results quality using the schema in [2].

Worker Selection. Since experience of the crowd workers highly impacts quality of crowdsourcing results, we use reputation-based worker selection [2]. Crowd workers are rated based on their contributions to CS tasks in most crowdsourcing platforms. Their reputation is based on these ratings. Reputation-based worker selection allows only crowd workers above a specified *reputation threshold* to select a CS task. In our experiments on the bespoke [17] crowdsourcing platform microWorkers.com[5], only workers from the "best workers" group participated.

Effective Task Preparation. The reverse engineering task has to be described clearly and unambiguously. The task design must keep the effort for fake contributions similar to correctly solving the task. This is known as *defensive design* [2]. In our experiment, crowd workers are provided with a brief description of the classification task and the available classifications with examples. At any step of the process, they can access this description.

The crowd worker view (cf. Fig. 3) displays the code fragment and references (cf. Sect. 3) with syntax highlighting, the available categories and a text input in which a brief explanation must be given to provide reasons for the classification. The minimal explanation length of 50 characters aims at reducing or at least slowing down fake contributions and allows for filtering during post-processing, e.g. filtering identically copied explanations. The *compensation policy* combines financial and non-financial rewards: For quality contributions, crowd workers receive a financial reward of 0.30 USD (platform average during the experiment) and a positive rating of their contribution as non-financial reward, adding to their reputation.

Ground Truth. To assess the quality of contributions by crowd worker, we employ the ground truth approach: Classification tasks with known correct answers form the ground truth. In this way, individual worker can be assessed based on the correctness of answers for these test questions. This information determines the *individual user score* $S(w_i) \in [0, 1]$ for each crowd worker $w_i \in W$ by comparing the amounts of correct classifications $C^+_{w_i}$ and false classifications $C^-_{w_i}$ as in Eq. (1). It can be used as weight factor during results aggregation.

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

Majority Consensus. We employ the majority consensus technique to aggregate the crowdsourcing results. For each code fragment to be classified, the classifications $C \subset W \times A$ are tuples (w_i, c_k) of a crowd worker $w_i \in W$ and the class $c_k \in A$ which the crowd worker selected. The resulting voting distribution $V : A \mapsto [0, 1]$ is calculated for all possible classes $c_i \in A$ as in Eq. (2):

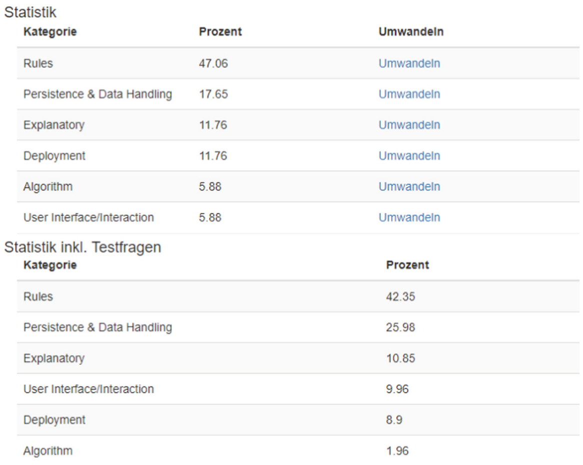
$$V(c_i) = \frac{\sum\limits_{(w,c) \in C | c = c_i} S(w)}{\sum\limits_{(w,c) \in C} S(w)} \tag{2}$$

5 https://microworkers.com/.

The aggregated result c^* of all crowd classifications is defined by the highest voting value as in Eq. (3):

$$c^* = \arg\max_{c \in A} V(c) \tag{3}$$

For more control, an overview with the result distributions and explanations was implemented, so that edge cases without clear majority can be found easily and decided (cf. Figs. 4 and 5).



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

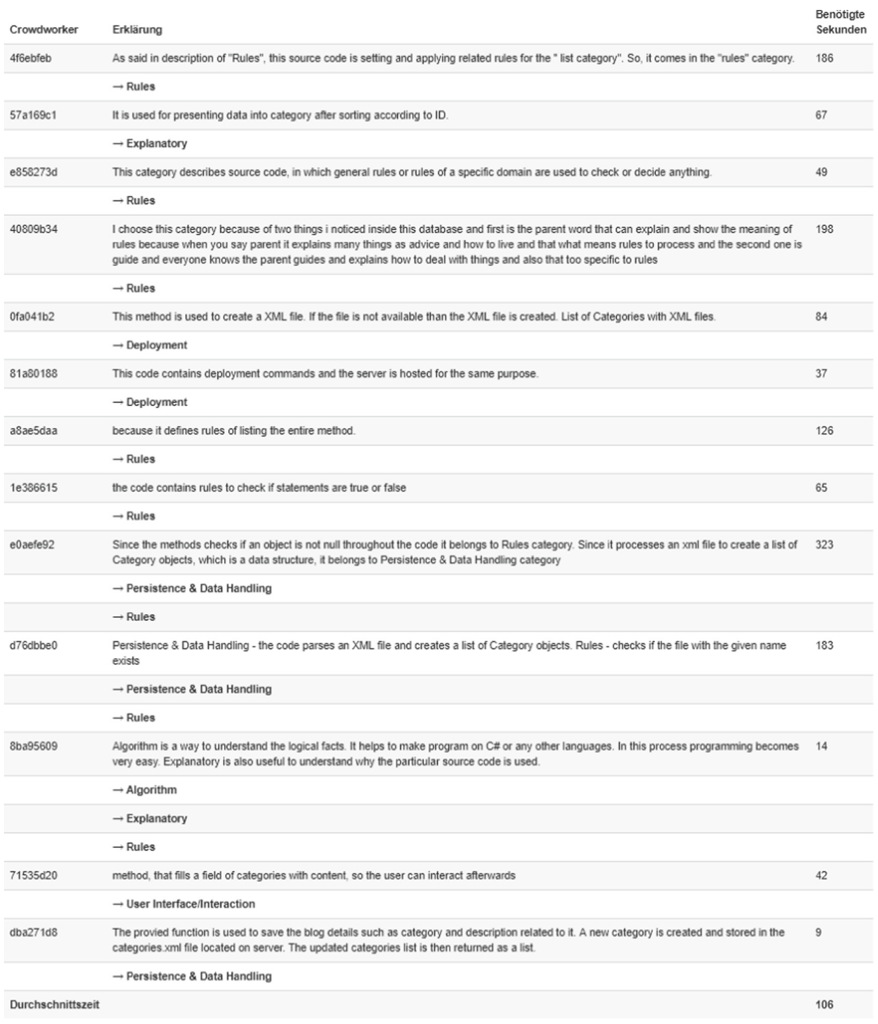
Fig. 4. Crowdsourcing results statistics view [10].

6 Related Work

In this section, we provide a brief overview on crowdsourcing research in reverse engineering and software engineering and position our CSRE approach accordingly (cf. Table 1).

6.1 Crowdsourcing in Reverse Engineering

Research applying crowdsourcing to reverse engineering is sparse. CrowdSource by Saxe et al. [23] is an approach for malware classification combining NLP with crowdsourcing. The initial data is provided by the crowd, the actual classification work, however, is performed using statistical NLP methods like full-text indexing and Bayesian networks. Based on the vast natural language corpus available on question and answer websites like StackExchange, CrowdSource creates a statistical model for malware capability detection. The model correlates low-level keywords like API symbols or registry keys with high-level malware capabilities like screencapture or network communication. In contrast to CSRE, CrowdSource follows a *passive crowdsourcing* [16] model: Crowdsourcing is employed only to generate the required input probabilities for the Bayesian model and not directly for performing the classification work.



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**

Fig. 5. Crowdsourcing result details view for migration engineer [10].

Table 1. Overview on existing crowdsourcing approaches in reverse engineering and software engineering.

Approach	Field	CS for	Act./pass.	CS Model
CrowdSource [23]	Malware classification	Initial text corpus	Passive	Sharing and reuse
CrowdDesign [19]	Component development	Programming	Passive, active	Peer production, microtasking
CrowdAdapt [20]	HCI	Adapting and evaluating layouts	Passive	Sharing and reuse
CrowdDesign [28]	HCI	Partial UI design	Active	Microtasking
(Stol 2014) [25]	Modeling, testing	Asset modeling, test automation	Active	Microtasking
(Satzger 2014) [22]	Software development	Software development	Active	Collaborative CS

6.2 Crowdsourcing in Software Engineering

Crowdsourcing has received wider consideration in software engineering. For instance, [19] presents a platform for crowd-supported creation of composite web applications. The web engineer creates the design of the web application as mashup based on information and interface components. Nebeling et al. combine a passive with an active crowdsourcing model: *Sharing and Reuse* is used in a community-based component library. Public components can be used by the web engineer to compose the web application. *Active crowdsourcing* is used for creating new components. The web engineer defines characteristics of the required component and posts an open call to a paid, external crowd. Improving the technical quality of the crowdsourced solution candidates is reported as one of the main issues. The survey in [17] provides a good overview on crowdsourcing in software engineering, showing increased research interest since 2010.

In the HCI field, crowdsourcing was successfully employed to adapt existing layouts to different screen sizes [20]. CrowdAdapt leverages the crowd for creating adapted web layouts and to select the best layout variants. It focuses on crowd-driven end-user development web layout tools. Crowdsourcing primarily serves as a means of exploring the design space and to elicit design requirements for various viewing conditions. CrowdAdapt, unlike other CS approaches in software engineering, uses unpaid crowd work. Unpaid crowd work can be successfully employed in HCI contexts due to the high number of users indirectly providing feedback through their interactions.

Similar to CSRE, CrowdDesign [28] employs the microtasking crowdsourcing model. To solve small user interface design problems, CrowdDesign uses paid crowd workers from Amazon Mechanical Turk. It focuses on diversity, i.e. for a set of decision points in the design space, CrowdDesign creates various and diverse solution alternatives. Early results report a high diversity, but only few crowd-created solutions achieved sufficiently high quality. In contrast, for CSRE, quality is the most relevant property. Diversity in the results is not intended.

Industrial case studies on CS in software development like [25] indicate that software development activities of lower complexity and relative independence are the most successful for CS. However, even more complex software development tasks can benefit from the lower costs, faster results creation and higher quality of successful crowdsourcing application. The case study considers two areas: test automation and front end modeling. Similar to CSRE, [25] focuses on the perspective of an enterprise crowdsourcing customer. Quality is one of the main problems as seen by a significant number of defects in the produced results. Stol et al. report on continuity problems since new crowd workers lack the experience from their predecessors and even re-introduce previously fixed bugs. From an enterprise perspective, they conclude that applicability of CS in software engineering is limited to self-contained areas without interdependencies, such as GUI design.

Latoza et al. [15] identify eight foundational orthogonal *dimensions of crowdsourcing for software engineering*: crowd size, task length, expertise demands, locus of control, incentives, task interdependence, task context and replication.

Existing successful crowdsourcing models like peer production, competitions and microtasking are characterize according to these dimensions. The CSRE Classification described in this article closely matches the microtasking model, as shown in Fig. 6. The differences are in only two of the eight dimensions: while in microtasking expertise demand is generally low, we consider it low to medium for source code classification. The amount of information about the entire system required by the worker (task context) is zero for microtasking, compared to low for the classification. The high similarity makes it likely 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 benefit from the high number of workers and the parallel execution of tasks. 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 [15]. CSRE meets both of these characteristics.

Distributed software development abstracting the workforce as crowd is described by Satzger et al. [22]. The public crowd is found on crowdsourcing platforms, the private crowd consists of company employees. The approach aims at collaborative crowdsourcing of software in enterprise contexts. It proposes to start with requirement descriptions in customer language. These are transformed into developer crowd tasks by a software architect. Developed collaboratively, the tasks are delegated to private and public crowds. Tasks can recursively be divided into smaller tasks and delegated to the crowd by crowd workers. The iterative development process tries to combine properties and artifacts from agile

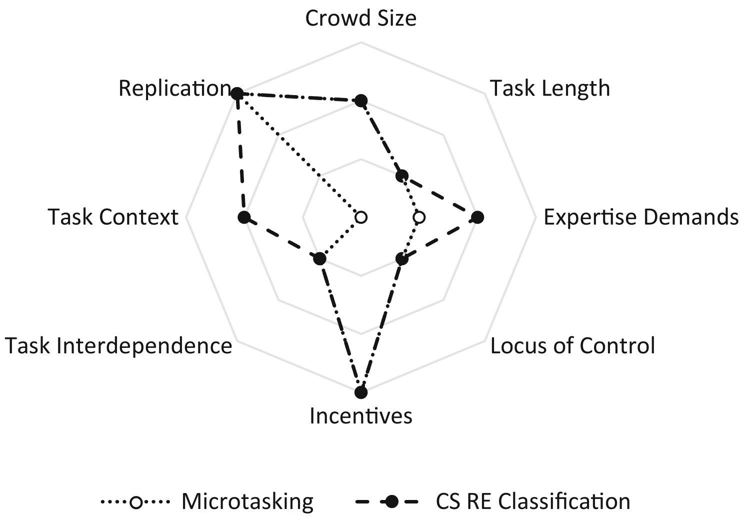


Fig. 6. Comparison of microtasking and crowdsourced reverse engineering (CS RE) classification [10].

development methodologies with collaborative crowdsourcing. Similarly, CSRE integrates with agile development, however, due to the nature of the reverse engineering classification task, it is not collaborative.

7 Evaluation

7.1 Experimental Design

We automatically extracted code fragments from BlogEngine.NET[6], an open-source ASP.NET-based blogging platform and randomly selected 10 of them. These 10 code fragments range from 7 to 57 LOC, on average 25.4 LOC. Using the following 8 categories from our source code knowledge ontology [11], we classified each of them manually:

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

Extending the 3 basic categories typically considered (presentation, application logic and persistence [7]), they provide a more fine-grained distinction of knowledge in source code. The implementation of CSRE was integrated into our existing source code annotation platform [11]. Crowd worker views with authentication mechanisms, classification task extraction based on doxygen (see footnote 3) and integration with crowdsourcing platform microWorkers were implemented.

Using `focus` and `blur` events, the crowd worker view tracks the time spent on it. The classification campaign ran for 14 days only with workers from the "best workers" group. 0.30 USD of financial reward were paid per 3 classifications.

7.2 Results

34 unique crowd workers contributed 187 classifications on our test data set. Table 2 shows the results. CF are the ten code fragments, Consensus indicates the result of the consensus voting. The numbers of classifications per category are stated in the categories cells. Bold values are the maxima, which are the basis for the majority consensus. Grey background marks the correct classification of the code fragment. Table 3 presents statistics to further analyze the results. $|C|$ is the number of classifications and $|W|$ the number of crowd workers. Note that the crowd worker and classification numbers differ due to multi-selections. The code fragment length in LOC is stated in l, Σt reports the overall time spent, $\bar{t}$ is the average time. Times are reported in seconds. The error rate f_e (cf. Eq. (4))

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

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

is the ratio of false classifications C^- to all classifications C of a code fragment. To investigate the degree of (dis-)agreement between the crowd worker classifications, we include entropy E (cf. Eq. 5)

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

and normalized Herfindahl dispersion measure H^* (cf. Eq. 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. E and H^* indicate the disorder/dispersion among crowd workers: a unanimous classification result yields $E = H^* = 0$. The higher the disagreement, the more different classifications, the closer E and H^* get to 1. Therefore, they are indicators of the classification certainty across the crowd workers. On average, 16 crowd workers created 18.7 classifications per code fragment.

7.3 Discussion

The average error rate of 0.655 seems high. With majority consensus, however, 7 of 10 code fragments were correctly classified. The minimum error rate was .25 on fragment B and the maximum 1 for fragment I. Provided a small expertise variation of the participating crowd workers, this indicates differences in the difficulty (fragment I was one of the longest) and the understanding of the categories. Rule was the most frequent classification (23.5%), Persistence & Data Handling (21.9%) s, Deployment the least voted (5.3%). No majorities

Table 2. Experimental results.

	Categories								
CF	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Consensus**
A	1	**14**	4	1	1	1	1	1	Algorithm
B	0	1	3	0	0	**12**	0	0	Rule
C	3	0	**4**	1	0	1	0	0	Persistence
D	2	5	**6**	3	0	5	2	0	Persistence
E	4	2	1	**9**	2	0	3	1	UIX
F	0	0	3	0	1	6	**7**	2	Config
G	3	1	1	2	2	**6**	0	0	Rule
H	0	2	**12**	2	4	3	2	2	Persistence
I	0	1	3	1	2	**8**	0	2	Rule
J	2	2	**4**	0	1	2	1	**4**	Persistence/Deployment

Table 3. Descriptive statistics.

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

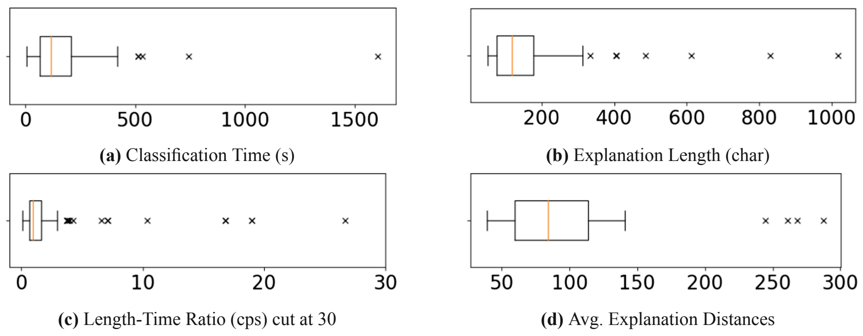
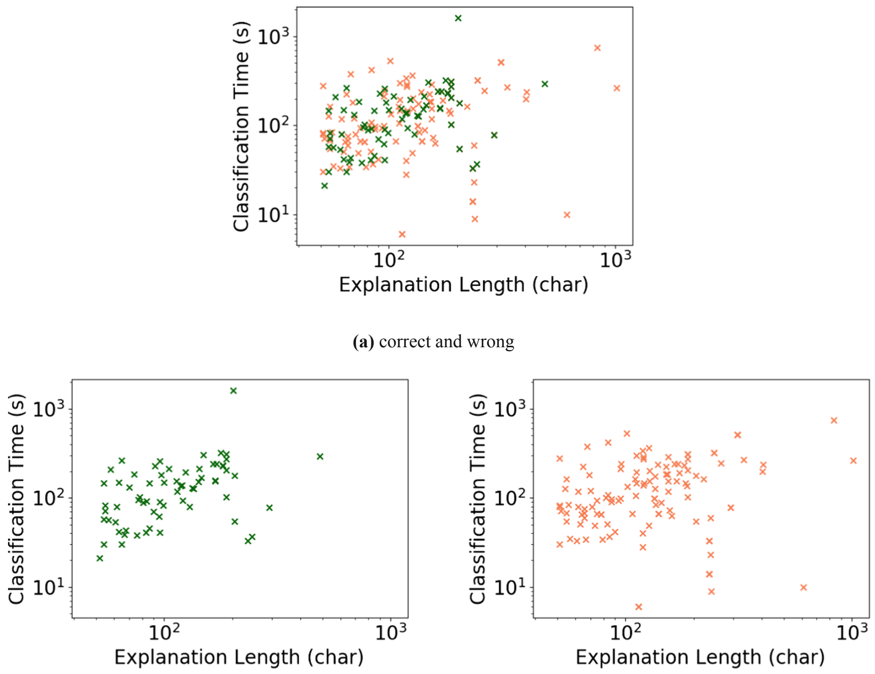


Fig. 7. Classification time and explanation length 1-dimensional distributions.

were achieved for Business Process and Explanatory, indicating that they might not be clear enough for the crowd workers. All other categories were correctly classified by the respective majorities.

Average entropy is 0.639 and average Herfindahl dispersion measure 0.757, their minima co-occur with minimal error rate, their maxima with the second-highest f_e. We found a significant ($\alpha = 0.05$) positive correlation (Pearson's $\rho = 0.724$, $p = 0.018$) between error rate and entropy and between error rate and Herfindahl dispersion measure ($\rho = 0.757$, $p = 0.011$), i.e. the more crowd workers vote one category, the less likely it is a wrong classification. No clear majorities for wrong classifications were observed. This supports the basic crowd-sourcing principle *"wisdom of the masses"* and the majority consensus assumption, that majorities are indicative of correct answers.

Our experiment did not show correlation between code fragment length and classification time (cf. also Fig. 8). This indicates influence of other variables and can be interpreted by assuming different levels of difficulty/clarity of the classification. Taking additionally correctness into account, correct classifications



(a) correct and wrong

(b) correct only

(c) wrong only

Fig. 8. Classification time and explanation length (log scaled). (Color figure online)

showed less time outliers[7] than wrong classifications. This can be interpreted that longer classification time or longer explanations are indicative of uncertainty, leading to wrong classifications in most observed cases. Most of the outliers in Fig. 8c were *Persistance & Data Handling* code fragments, which was the second most frequently voted category and indicating, that the formulation of this category is not clear enough.

To further analyze the work of the crowd workers, we consider the distributions of time and explanation length, as well as derived length-time ratio and average explanation similarity per worker as shown in Fig. 7. Time median was 117.5 s, but the observed times varied widely: the inter quartile range was IQR = 143.25 s while upper outliers reached almost half an hour (1606 s). With the lower quartile at 65 s and min time 6 s, the relatively low times show that the tasks were formulated appropriately for microtasking, but also point to fake contributions as described below. Longer times, however, do not imply better accuracy as all but one time outlier in Fig. 7a belonged to C^-. Results of one specific crowd worker showed suspiciously identical time measurements (3×14 s, 3×33 s, 3×515 s), with identical explanations in all three groups which most likely resulted from the use of a record-and-replay script.

[7] We use $Q3 + 1.5\text{IQR}$ as outlier threshold in this article.

Explanation lengths (cf. Fig. 7b) ranged between 51 and 1017 characters (median $\tilde{d} = 119$) and were relatively close (IQR = 100.5) to the min threshold of 50 chars. Also, the lower quartile of 76.25 chars and outliers for time and length only appearing above the third quartile show that most of the workers wrote rather short explanations in short time. Figure 8 shows explanation length and time in relation, with correct classifications in green and wrong in red.

Time and explanation length distributions revealed that some crowd workers tried to gain the reward quickly through fake contributions. To identify these unlikely fast classifications, we calculated the length-time ratio in characters per second (cps). As shown in Fig. 7c, within a range of 0.13 to 61.2 cps, most of the values are distributed very closely (IQR 0.96 cps) around a median of 0.94 cps. The 20 outliers are likely to have copied texts. Analysis of the contents showed that these copies were most often copies of own previous explanations and sometimes from the task description. Since time does not only include writing time for justifications, but also time for source reading, understanding, deciding and selecting the classification, very high length-time ratios were only reachable if also little time was spent for thorough consideration, resulting in only 4 of 20 outliers belonging to C^+. Even when assuming fast thinking and typing capabilities, speeds are not likely to exceed the upper level of 16 cps measured at competitions[8]. However, 7 crowd workers exceeded this level. Manual analysis of explanations of the outliers determined the fastest worker who did not copy text and classified correctly at approx. 6.6 cps. The upper quartile at 1.63 cps shows that the vast majority of workers produced results in reasonable time.

To further identify workers who tried to complete the tasks very quickly through copying, we calculated average similarity of explanations per user using pairwise Levenshtein distance. The average distances range from 39 to 287 and are concentrated (IQR = 54.5) around a median of 83.8 (cf. Fig. 7d). Identical copies (distance 0) were received from 10 of 34 (29.4%) crowd workers, but the min average of 39 indicates that crowd workers did not exclusively copy. Note that 7 workers provided only one classification and could thus not copy their own texts. When normalized with the average explanation length, two workers had even less than 50% relative changes, copying most explanations with only minor adaptions. Manual inspection of crowd worker responses furthermore showed that 3 of 34 replied exclusively with code in their explanations, indicating a wrong understanding of the task.

In contrast to these negative cases, we also observed very thorough workers: Fragment J was split-voted as Persistence/Deployment. The explanation texts argued that J is related to persistence because the fragment is part of a class related to persistence. This observation was very interesting, because our dataset did not include the entire class. Thus, several crowd workers looked up the sample source code on the internet and read also the surrounding parts in order to classify. This level of active engagement and investment of time by the crowd workers to complete their task positively surprised us.

[8] cf. http://www.intersteno.org/.

In spite of the cases of low quality and fake contributions reported above – which are a known characteristic of crowdsourcing [2] – our quality control measures proved robust enough to yield 70% overall correctness. Our experiment has shown that 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 Conclusions and Future Work

This article motivated research in crowdsourced reverse engineering and outlined three major challenges – (1) automatic task extraction, (2) source code anonymization and (3) results aggregation and quality control – for applying crowdsourcing in the reverse engineering domain. To illustrate these challenges, we presented CSRE, our approach for concept assignment based on crowdsourced classifications. Extending our previous work in [10], we provided a detailed theoretical basis of the reverse engineering problem of concept assignment and, following an overview on existing approaches, presented a re-formulation of concept assignment as classification problem. Figure 2 refines the CSRE process [10], introducing relevant activities and artifacts and specified the anonymization algorithm in Algorithm 1. We presented a detailed problem analysis and the CSRE approach to address each of the challenges. Our classification task extraction method re-uses existing software documentation tools. To address aggregation and quality of crowdsourced results, we showcased a method combining several crowdsourcing quality control techniques. The extended review of related work of crowdsourcing in software engineering and reverse engineering shows a lack of crowdsourcing consideration in reverse engineering. At the same time, we reported examples of successful application of crowdsourcing in software engineering and demonstrated the similarity of crowdsourced concept assignment to microtasking in eight dimensions.

We reported on our experiences from an evaluation experiment on the crowdsourcing platform microWorkers, which produced 187 results by 34 crowd workers, classifying 10 code fragments at a low cost. The results' quality indicates that crowdsourcing is a suitable approach for certain reverse engineering activities. We were positively surprised by observations demonstrating an unexpectedly high level of engagement and effort by individual crowd workers to solve the tasks correctly. Calculating entropy and Herfindahl dispersion measure, we could see some evidence for the applicability of the crowdsourcing principle "wisdom of the masses" in our context, since higher levels of agreement was indicative of correctness. Extending previous evaluations [10], we added a focus on crowd worker behavior, in particular traces of fake contributions. The detailed analysis includes definitions, figures and interpretations of time and length distributions, time-length ratio and levenshtein-based similarity.

Future research challenges include achieving similar results in other areas of reverse engineering and improving the quality of the results. To identify these further reverse engineering activities and corresponding crowdsourcing paradigms, the matching procedure we used in Sect. 6 can be employed. An evaluation with a larger budget and crowd worker base should yield more insights into the applicability of crowdsourcing for reverse engineering activities, especially when combined with more specific, tailored measures of agreement in crowd worker results. An interesting research challenge is the crowd-based specification of concrete problem and solution domain models. Further investigations will show if this is possible through isolated microtasking with a more comprehensive classification ontology specific to the legacy system instance, or whether complex collaborative crowdsourcing approaches are necessary. While anonymization has been shown as the most difficult challenge, providing many opportunities for further research, application of our proposed method in contexts without anonymization requirements like intra-organization settings or open source projects will produce further insights.

Acknowledgements. The authors would like to thank Felix Förster for his valuable contributions towards this research. This research was supported by the eHealth Research Laboratory funded by medatixx GmbH & Co. KG.

References

1. Abebe, S.L., Tonella, P.: Extraction of domain concepts from the source code. Sci. Comput. Program. **98**, 680–706 (2015). https://doi.org/10.1016/j.scico.2014.09.012
2. Allahbakhsh, M., Benatallah, B., Ignjatovic, A., Motahari-Nezhad, H.R., Bertino, E., Dustdar, S.: Quality control in crowdsourcing systems: issues and directions. IEEE Internet Comput. **17**(2), 76–81 (2013). https://doi.org/10.1109/MIC.2013.20
3. Arboit, G.: A method for watermarking Java programs via opaque predicates. In: The Fifth International Conference on Electronic Commerce Research (ICECR-5), pp. 102–110 (2002)
4. Aversano, L., Canfora, G., Cimitile, A., De Lucia, A.: Migrating legacy systems to the web: an experience report. In: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, pp. 148–157. IEEE Computer Society Press (2001)
5. Biggerstaff, T.J., Mitbander, B.G., Webster, D.E.: Program understanding and the concept assignment problem. Commun. ACM **37**(5), 72–82 (1994). https://doi.org/10.1145/175290.175300

6. Biggerstaff, T., Mitbander, B., Webster, D.: The concept assignment problem in program understanding. In: Proceedings of the 15th International Conference on Software Engineering, ICSE 1993, pp. 482–498. IEEE Computer Society Press (1993). https://doi.org/10.1109/ICSE.1993.346017
7. Canfora, G., Cimitile, A., De Lucia, A., Di Lucca, G.A.: Decomposing legacy programs: a first step towards migrating to client-server platforms. J. Syst. Softw. **54**(2), 99–110 (2000). https://doi.org/10.1016/S0164-1212(00)00030-3
8. Ceccato, M., Di Penta, M., Falcarin, P., Ricca, F., Torchiano, M., Tonella, P.: A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. Empir. Softw. Eng. **19**(4), 1040–1074 (2014). https://doi.org/10.1007/s10664-013-9248-x
9. Eaddy, M., et al.: Do crosscutting concerns cause defects? IEEE Trans. Softw. Eng. **34**(4), 497–515 (2008). https://doi.org/10.1109/TSE.2008.36
10. Heil, S., Felix, F., Gaedke, M.: Exploring crowdsourced reverse engineering. In: Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering, pp. 147–158. SCITEPRESS - Science and and Technology Publications (2018)
11. Heil, S., Gaedke, M.: AWSM - Agile web migration for SMEs. In: Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering, pp. 189–194. SCITEPRESS - Science and Technology Publications (2016). https://doi.org/10.5220/0005869301890194
12. Heil, S., Gaedke, M.: Web migration - a survey considering the SME perspective. In: Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering, pp. 255–262. SCITEPRESS - Science and Technology Publications (2017). https://doi.org/10.5220/0006353502550262
13. Heil, S., Siegert, V., Gaedke, M.: ReWaMP: rapid web migration prototyping leveraging webAssembly. In: Mikkonen, T., Klamma, R., Hernández, J. (eds.) ICWE 2018. LNCS, vol. 10845, pp. 84–92. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91662-0_6
14. Kazman, R., Brien, L.O., Verhoef, C.: Architecture reconstruction guidelines third edition. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 2003
15. Latoza, T.T.D., van der Hoek, A.: Crowdsourcing in software engineering: models, opportunities, and challenges. IEEE Softw. **33**(1), 1–13 (2016)
16. Loukis, E., Charalabidis, Y.: Active and passive crowdsourcing in government. In: Janssen, M., Wimmer, M.A., Deljoo, A. (eds.) Policy Practice and Digital Science. PAIT, vol. 10, pp. 261–289. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-12784-2_12
17. Mao, K., Capra, L., Harman, M., Jia, Y.: A survey of the use of crowdsourcing in software engineering. J. Syst. Softw. **126**, 57–84 (2017). https://doi.org/10.1016/j.jss.2016.09.015
18. Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.I.: An information retrieval approach to concept location in source code. In: 2004 Proceedings of 11th Working Conference on Reverse Engineering, pp. 214–223. IEEE (2004). https://doi.org/10.1109/WCRE.2004.10
19. Nebeling, M., Leone, S., Norrie, M.C.: Crowdsourced web engineering and design. In: Brambilla, M., Tokuda, T., Tolksdorf, R. (eds.) ICWE 2012. LNCS, vol. 7387, pp. 31–45. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31753-8_3

20. Nebeling, M., Speicher, M., Norrie, M.: CrowdAdapt: enabling crowdsourced web page adaptation for individual viewing conditions and preferences. In: Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing System, pp. 23–32 (2013). https://doi.org/10.1145/2480296.2480304
21. Rose, J., Jones, M., Furneaux, B.: An integrated model of innovation drivers for smaller software firms. Inf. Manag. **53**(3), 307–323 (2016). https://doi.org/10.1016/j.im.2015.10.005
22. Satzger, B., et al.: Toward collaborative software engineering leveraging the crowd. In: Economics-Driven Software Architecture, pp. 159–182. Elsevier (2014). https://doi.org/10.1016/B978-0-12-410464-8.00008-8
23. Saxe, J., Turner, R., Blokhin, K.: 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), pp. 68–75. IEEE, October 2014. https://doi.org/10.1109/MALWARE.2014.6999417
24. Shepherd, D., Fry, Z.P., Hill, E., Pollock, L., Vijay-Shanker, K.: Using natural language program analysis to locate and understand action-oriented concerns. In: Proceedings of the 6th International Conference on Aspect-Oriented Software Development - AOSD 2007, p. 212. ACM Press, New York (2007). https://doi.org/10.1145/1218563.1218587
25. Stol, K.J., Fitzgerald, B.: 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, pp. 187–198. ACM Press, New York (2014). https://doi.org/10.1145/2568225.2568249
26. Wagner, C.: Model-Driven Software Migration: A Methodology. Springer, Wiesbaden (2014). https://doi.org/10.1007/978-3-658-05270-6
27. Warren, I.: The Renaissance of Legacy Systems: Method Support for Software-System Evolution. Springer, Heidelberg (2012). https://doi.org/10.1007/978-1-4471-0817-7
28. Weidema, E.R.Q., López, C., Nayebaziz, S., Spanghero, F., van der Hoek, A.: Toward microtask crowdsourcing software design work. In: Proceedings of the 3rd International Workshop on CrowdSourcing in Software Engineering - CSI-SE 2016, pp. 41–44. ACM Press, New York (2016). https://doi.org/10.1145/2897659.2897664