A Mechanism for Automatically Extracting Reusable and Maintainable Code Idioms from Software Repositories

Argyrios Papoudakis[a], Thomas Karanikiotis[b] and Andreas L. Symeonidis[c]
Dept. of Electrical and Computer Eng., Aristotle University of Thessaloniki, Thessaloniki, Greece

Keywords: Code Idioms, Syntactic Fragment, Software Reusability, Software Maintainability, Software Engineering.

Abstract: The importance of correct, qualitative and evolvable code is non-negotiable, when considering the maintainability potential of software. At the same time, the deluge of software residing in code hosting platforms has led to a new component-based software development paradigm, where reuse of suitable software components/ snippets is important for software projects to be implemented as fast as possible. However, ensuring acceptable quality that will guarantee basic maintainability is also required. A condition for acceptable software reusability and maintainability is the use of idiomatic code, based on syntactic fragments that recur frequently across software projects and are characterized by high quality. In this work, we present a mechanism that employs the top repositories from GitHub in order to automatically identify reusable and maintainable code idioms. By extracting the Abstract Syntax Tree representation of each project we group code snippets that appear to have similar structural and semantic information. Preliminary evaluation of our methodology indicates that our approach can identify commonly used, reusable and maintainable code idioms that can be effectively given as actionable recommendations to the developers.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Delivering software fast and with good quality has already been a quest for software engineering practitioners. This quest has become more evident in our era of rapid software prototyping and the establishment of the agile, component-based software engineering paradigm. The narrow time-to-market schedules and the constantly changing features undermine the quality of developed software and threaten the maintainability of software projects. Abran et al. (Abran and Nguyenkim, 1993) argue that the software maintaining process consumes the largest percentage of software costs. A poor software maintainability "score" usually leads to increased bug fixing times and inability to properly plan features development, this way derailing the software project time frame, both in time as well as in budget context.

In order to speed up software development, reduce programming effort and minimize the risks of defective software, developers constantly try to reuse existing software components/libraries/code snippets, benefiting from software that has already been implemented and released and resides in some code hosting facility. However, the quality of the components to be reused is not granted and suboptimal searches may lead to code snippets of poor quality that are tough to integrate and could possibly introduce bugs. So, there is a high need for software components that the developers can easily reuse and integrate into their software and that can guarantee acceptable quality.

This requirement can be satisfied by using idiomatic code or code idioms. Code idioms are small syntactic fragments that recur frequently across various software projects, with simple tasks to execute. According to Allamanis et al. (Allamanis and Sutton, 2014), code idioms appear to be significantly different from "previous notions of textual patterns in software", as they involve syntactic constructs (decision-making statements, loops and exception-handling blocks). Idioms, which can improve the overall quality of the software they are used into, are characterized by high reusability and can make the code significantly easier to maintain (Hnatkowska and Jaszczak, 2014). Popular Integrated Development Environments (IDEs) already support idioms for various programming languages. However, there are only few approaches that aspire to mine code idioms automat-

[a] https://orcid.org/0000-0001-7371-2277
[b] https://orcid.org/0000-0001-6117-8222
[c] https://orcid.org/0000-0003-0235-6046

Papoudakis, A., Karanikiotis, T. and Symeonidis, A.
A Mechanism for Automatically Extracting Reusable and Maintainable Code Idioms from Software Repositories.
DOI: 10.5220/0011279300003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 79-90
ISBN: 978-989-758-588-3; ISSN: 2184-2833
Copyright © 2022 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

ically (Allamanis and Sutton, 2014). The few existing ones are mostly based on statistical methods and are not able to capture idioms based also on their increased usage among top-level and popular repositories.

In this work, we present a mechanism that can harness data from the top-level, most popular open-source projects, residing in the code hosting platform GitHub, and identify reusable and maintainable code idioms, in a completely unsupervised manner. This mechanism extracts the most-used code blocks found across different projects, compares both their structural and their semantic information and groups code snippets with similar architecture and functionality. After automatically processing the generated clusters and discarding the ones that do not meet specific requirements, our system produces a set of code idioms that have been used considerably across top-level projects. To sum up, the main contributions of our approach lie in the following axes:

- We present an approach for automatically identifying and extracting code idioms used in various projects.
- We build our methodology on projects of high maintainability and reusability, which is shown in the projects' stars and forks.
- The extraction of code idioms is separated and accomplished differently for the various code blocks found in programming languages, such as *if* and *for*, which provide different functionalities.
- We provide an abstract form of the final idioms extracted by our system, which a developer can easily integrate into his/her project and modify accordingly.
- Having evaluated our approach and examined the extracted idioms, we identify that these idioms are used in various important programming tasks, such as null checking, looping through the elements of an array and reading the content of an object or file. Such tasks considerably affect software maintainability.

The rest of this paper is organized as follows. Section 2 reviews the current approaches in the literature that aspire to extract small fragments of code that recur frequently across projects and are characterized by high quality. Our methodology, along with the dataset we have created, the processing steps and the models we have employed are depicted in Section 3, while in Section 4 we evaluate the code idioms that were extracted by our approach. In Section 5, we analyze potential threats to the internal and external validity of our approach and, finally in Section 6 we provide insight for further research and conclude the paper.

2 RELATED WORK

The importance of software maintainability and reusability has gained increasing interest during the recent years, where the time schedules for projects are very tight, features are being constantly added or modified and the maintenance costs are rising. To address these issues, the component-based software paradigm, which is easy to debug, maintain and reuse, has gained ground. This paradigm relies on software components and small fragments of code, that execute specific and well-defined programming tasks and can be easily understood by the developers. Idiomatic code or code idioms are such small syntactic blocks that recur frequently across different projects and execute simple tasks. Idioms can significantly improve the software maintainability of a project, while they provide small, compact and highly reusable code blocks, which the developers can effortlessly comprehend and (re)use.

The process of extracting useful information from semi-structured data is a well-known task, in which the developers attempt to detect frequently encountered components. Even though mining frequent patterns is an important task, it is not directly correlated to the idioms mining challenge. A problem that is highly connected to idioms mining is the identification of code clones. Code clones aim to detect similar blocks of code across different projects. Zhang and Wang (Zhang and Wang, 2021) proposed a tool, named *CCEyes*, which is based on semantic vector representations of big repositories and can identify similar code fragments. Upon evaluating their approach, the authors argue that *CCEyes* outperforms state-of-the-art approaches in code cloning. Similarly, Ji et al. (Ji et al., 2021) are mainly focused on the hierarchical structure of the Abstract Syntax Tree, where they apply an attention mechanism, in order to examine the importance of different tree nodes. The results of the experiments conducted by the authors show that their proposed methodology achieves superior results compared to the baseline methods.

Another area related to idioms mining is the API mining problem. In this task, the main goal is the extraction of sequences or graphs of API method calls. While there have been a number of approaches that aspire to mine API usage patterns, Wang et al. (Wang et al., 2013) argued that they lack appropriate metrics to evaluate the quality of their results. Thus, they proposed two metrics, called *succinctness* and *coverage* in order to measure the quality of the mined usage patterns of API methods from the view of the developers. At the same time, they proposed *UP-Miner*, a system that mines API usage patterns from source code based on the similarity of the sequence and a clustering algorithm. The experiments conducted on the effectiveness of *UP-Miner* indicate that the *UP-Miner* outperforms the existing approaches. Moreover, Fowkes and Sutton (Fowkes and Sutton, 2016) aspire to address the limitation of the large parameter tuning that is required in the most of the API mining algorithms, by proposing the *Probabilistic API Miner - PAM*, which is a near parameter-free probabilistic algorithm for mining API call patterns. *PAM* outperformed previously proposed systems, while the authors argue also that the most of the existing API calls are not well documented, which makes the API usage patterns mining a demanding task.

While all these approaches are closely related to idioms mining, significant variations exist. Code clones aspire to detect similar code that executes the same functionality, but not identical one, which is prerequisite for the idioms mining. Similarly, API mining is interested only in sequences or graphs mining and not on the code itself. Additionally, while code search approaches cover a much wider area, they are usually not focused on code quality and maintainability.

When it comes to idiom mining, research approaches try to model the problem from a statistical point of view. Allamanis and Sutton (Allamanis and Sutton, 2014) proposed *HAGGIS*, a system for mining code idioms, which is mostly based on nonparametric probabilistic tree substitution grammars. Upon applying *HAGGIS* to some of the most popular repositories, the authors claim that the extracted idioms are being used in a wide range of projects. However, the most of the idioms that *HAGGIS* was able to identify perform only easy tasks, such as object creation and exception handling. In their next approach, Allamanis et al. (Allamanis et al., 2018) focused only on loop idioms. They were based again on probabilistic tree substitution grammars, which they augmented with important semantic, in order to identify loop blocks that recur frequently and meet the basic requirements of code idioms. The identified loop idioms seem to appear frequently, while they are able to identify opportunities for new code features. However, a study conducted in 2019 by Tanaka et al. (Tanaka et al., 2019) indicated that a lot of the mined code idioms are not being used frequently, while some types of idioms, such as the *Stream* idioms that operate on the elements of a collection, are only used at times. At the same time, the lack of complete idioms or examples leads to bad idioms usage. Finally, Sivaraman et al. (Sivaraman et al., 2021) proposed Jezero, a system that uses nonparametric Bayesian methods, in order to extract canonicalized dataflow trees. Even though Jezero performs better than other baseline approaches, it appears to significantly depend on the extent and the nature of the code it is applied to.

In this work, we aspire to tackle the main drawbacks that are introduced in the approaches found in the literature. We propose a mechanism to extract idioms from the most popular and reused projects from GitHub, characterized by high maintainability and reusability (Papamichail et al., 2016a), split the software into small meaningful code snippets and group commonly used code blocks across different projects that have similar structural and semantic information. These groups are then filtered to discard code snippets that do not meet basic requirements of the idiomatic code and the final set of code idioms is generated. We argue that code idioms can effectively help the developers during both the development phase, where components that accomplish simple tasks are selected to be reused, and the maintenance phase. At the same time, we argue that the extraction of the code idioms from repositories that have a big number of stars (users that can comprehend the main content and functionality of the project) and forks (users that use the whole project or project components), along with their long lifespan, is a good indicator of idiomatic code.

3 METHODOLOGY

In this section we present the architecture of our code idioms mining system (shown in Figure 1).

3.1 Dataset Construction

The first step towards creating our system is to generate a dataset that comprises of code snippets derived from the most popular software projects. The programming language selected for creating our corpus is *Java*, as it is a well-structured language and is considered ideal for mining code patterns.

In order to build our dataset, we use the 1,500 most popular Java repositories from *GitHub*[1], based on their reputation with respect to their number of stars and forks and their long lifespan, as it originates from the date of their first release. As our main target is the identification of reusable and maintainable code idioms, the initial dataset used in our system needs to

[1]https://github.com/

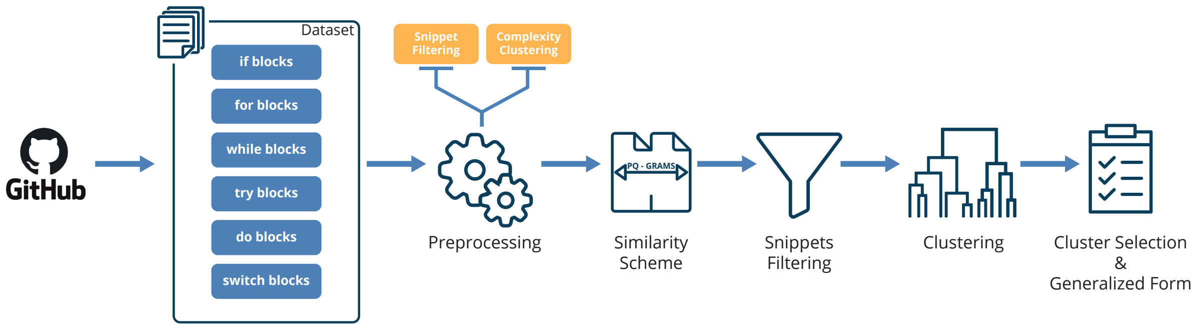

Figure 1: Overview of the Code Idioms Mining System.

include idiomatic source code that is popular among the developers. This can be ensured by the reputation metrics; forks measure how many times a software repository has been cloned and, thus, a large number of forks indicates great reusability, while the number of stars reflects the attractiveness of the project and hence a qualitative and maintainable software. Each project should also satisfy specific criteria posed by the idioms mining problem and the need for highly maintainable and reusable code. Except from the high number of stars and forks, each project needs to exhibit a large number of different contributors as well as steady and short release cycles (i.e. new features constantly added). These requirements aspire to ensure maintainability and reusability, as they are reflected by the project's persistence in time and its high acceptance from a lot of developers.

From the 1,500 software repositories, we randomly select 1,000 repositories to be used in the initial stage of our approach, where the code idioms are identified, and employ the remaining 500 repositories to evaluate the extracted idioms. The source code of these projects needs to be converted into a form suitable for our models. For this purpose, we make use of the *Abstract Syntax Tree (AST)* representation, which maintains both the structural and the semantic information of the source code. Specifically, the execution order of the code statements is encoded in the structure of the AST, while the semantic information is stored in the leaves of the tree, where the names of variables, methods and objects are found. The transformation of the source code files to ASTs is performed with the use of the *ASTExtractor*[2] tool.

As the code idioms are small syntactic fragments with a single semantic role and execute specific and well-defined programming tasks, we can easily assume that the most important code idioms concern a small block of code (e.g. an *if* block or a *for* block) and not a sequential snippet of code. Therefore, we mainly focus on fragments of code that belong to the Control Flow Statements (CFSs), which break up the sequential execution of the code with looping, decision-making or exception-handling statements. The main CFSs used in nearly any programming language are the *If*, *For*, *Try*, *While*, *Do* and *Switch* blocks. After traversing the ASTs of the source code files, we extract all the fragments of code that belong to the CFSs categories. Each CFS category is handled independently, as the snippets of each category have totally different semantic content and there is no similarity between them, that could lead to the identification of a code idiom. Table 1 depicts the number of snippets that each category of the CFS contains, where the *Enhanced For* statements refer to the iteration on the elements of an array or collection. It should be noted that the number of the *If* Statements was too large, so we narrowed it down using only 300 repositories and split them into 7 different files. The data have been publicly available and can be accessed at Zenodo[3]. The data can be accessed by any programming language, as it completely follows the csv standards.

Table 1: Number of Snippets per Category.

Type of CFS Blocks	# Snippets
If	4,988,452
For	411,770
Enhanced For	495,256
Try	716,776
While	174,644
Switch	106,904
Do	10,739

[2]https://github.com/thdiaman/ASTExtractor

[3]https://doi.org/10.5281/zenodo.5391965

3.2 Preprocessing

Prior to calculating the similarity matrix between code snippets, we have to adjust the initially created dataset to the requirements of the mining idioms problem. Taking into account the large number of repositories we employed, it is crucial that we reduce the computational cost, while, at the same time, discard data that may lead to deficient results. Therefore, we apply some preprocessing steps, which make the implementation of the main clustering procedure feasible.

As it has been already mentioned, the code idioms are small syntactic fragments with specific and well-defined programming tasks to execute. Thus, a larger snippet of code would not be part of a code idiom, as it contradicts with the main principles of code idioms. In order to exclude larger blocks of code from our dataset, we discard all the snippets that are greater than seven logical lines of code, i.e. executable lines of code, as idioms need to be significantly small and execute simple tasks and they usually do not exceed the above threshold [4][5]. It should be mentioned, however, that this is just a tunable parameter in our methodology and can easily be changed for further experimentation. Table 2 depicts the number of snippets each category contains, after removing the large snippets of code.

Table 2: Number of Snippets per Category after Filtering.

Type of CFS Blocks	# Snippets
If	1,049,729
For	194,299
Enhanced For	252,587
Try	246,632
While	55,438
Switch	10,482
Do	3,482

From the Table 2 we can notice that the first four categories (i.e. *If*, *For*, *Enhanced For* and *Try* statements) include a large amount of code snippets, which greatly impedes the effective clustering analysis of the next steps, especially during the calculation of the similarity matrices. In order to cope with this limitation, we split the respective categories into smaller parts, applying an initial clustering. The main goal of this initial clustering is the ability to perform only targeted comparisons when applying the main clustering, avoiding the comparisons between code blocks that are considerably different.

[4]https://programming-idioms.org/

[5]https://www.nayuki.io/page/good-java-idioms

The initial clustering applied on this step of our modelling procedure is based on the code complexity of the snippets. Snippets with a quite large difference in their complexity probably perform different tasks, while, even if they execute the same programming task, they are based on different approaches. Thus, we apply a clustering algorithm making use of two complexity metrics; the *McCabe's Cyclomatic Complexity* (McCabe, 1976), which counts the independent execution paths of the source code, as well as the *total number of variables*, *methods* and *objects* that the code contains. These two metrics are used as features for the clustering algorithm, in order to create groups that contain snippets with the same characteristics.

Applying this preprocessing step in the sets of *If*, *For*, *Enhanced For* and *Try* statements, we create several subsets that contain a significantly smaller number of snippets. The main point of this modelling step is that each of our sets contains only fragments of code that appear to have similar code complexity and hence they could lead to the identification of a code idiom. Table 3 depicts a summary of the results of the initial clustering, including the number of different groups that are created and the average number of snippets that are included in each group.

Table 3: Initial Clustering Results.

Type of CFS Blocks	# Groups	Average # Snippets
If	21	49,987
For	8	24,287
Enhanced For	10	25,258
Try	10	24,663

3.3 Similarity Scheme

Before applying the main clustering algorithm, we need to calculate a distance matrix for each one of the subsets of our corpus, which reflects the similarity between two snippets of code. For the comparison of the snippets, we use their AST representations and calculate the *Tree Edit Distance (TED)*. TED is defined as the minimum cost sequence of edit operations (node insertion, node deletion and label change) that can transform one tree into another (Tai, 1979). Over the years, various methods have been introduced to calculate the TED, in order to improve the runtime algorithm's complexity (Zhang and Shasha, 1989; Klein, 1998). It is remarkable, though, that the complexity in all these implementations is greater or equal to O(n^2). In order to avoid using such a computationally expensive method, we approximate the TED by using the pq-Grams algorithm (Augsten et al., 2005).

Towards using the pq-Grams algorithm, we first need to transform each AST into an ordered label tree T, using the type of AST statements as labels and connecting the nodes so that the tree is traversed in a preorder manner. Then a *pq-Extended-Tree* T^{pq} is constructed by adding null nodes on the tree T. Precisely, $p-1$ ancestors are added to the root of the tree, $q-1$ children are added before the first and after the last child of each non-leaf node and q children are inserted to each leaf of T. In our methodology, we define $p=2$ and $q=3$, as these values have been used significantly in tree matching approaches with great results. Figure 2 illustrates the extended tree transformation of a simple labelled tree, where $p=2$ and $q=3$.

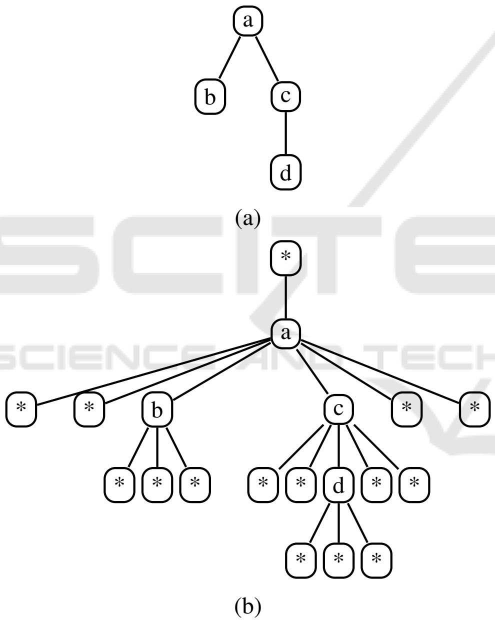

Figure 2: (a) Example tree T. (b) $T^{2,3}$ Extended-Tree.

For each Extended-Tree, we need to calculate a list of all the pq-Gram patterns it contains. A pq-Gram pattern is defined as a subtree of the extended tree T^{pq} that consists of an anchor node with $p-1$ ancestors and q children. These lists are called *Profiles* $P(T)$ and are used to calculate the distance between trees. For instance, the profile of the tree in Figure 2 is the list $[{*}a{*}{*}b, {*}a{*}bc, {*}abc{*}, {*}ac{*}{*}, ab{*}{*}{*}, ac{*}{*}d, ac{*}d{*}, acd{*}{*}, cd{*}{*}{*}]$. Overall, the pq-Gram distance between two trees T_1 and T_2 is defined as follows:

$$distance(T_1,T_2) = 1 - 2 * \frac{|P^{p,q}(T_1) \cap P^{p,q}(T_2)|}{|P^{p,q}(T_1) \cup P^{p,q}(T_2)|} \quad (1)$$

As it is depicted in Equation 1, the distance between two trees depends on the number of mutual pq-Grams patterns contained in both their profiles divided by the union of the two lists, which results in a value between 0 and 0.5 (when the union contains double the instances of the intersection) (Augsten et al., 2008). It is obvious that the similarity between two trees can be easily calculated using the formula $1 - distance(T_1,T_2)$. The final similarity value ranges from 0 to 1.

3.4 Snippets Filtering

The pq-Gram algorithm discussed in the previous section outputs the distance matrices calculated between code snippets in the same subsets presented above. We, then, proceed to the next step of our modelling procedure, where we remove duplicate instances from the data. The term duplicate instances refers to two different cases:

- snippets with zero pq-Gram distance that belong to the same code repository, as it is quite common for a fragment of code to be used numerous times in the same repository.
- snippets with zero pq-Gram distance that derived from different code repositories but, however, originate from the same package or library. It is a usual case, when developers need to make a change to an existing library or package and, thus, they include it into their project.

It should be noted that this processing step, which removes the duplicate instances from the data, is important in order to ensure that the resulting clusters will contain patterns used by many different software projects and, therefore, many different developers.

3.5 Clustering

In order to group code snippets, we perform *Agglomerative Hierarchical Clustering*, which is a bottom-up approach that initially considers each snippet as a separate cluster and iteratively merges the groups with the lesser distance. As a method of measuring the distance between clusters we use the *average linkage*, which is defined as the average distance between all the snippets, while the optimal number of clusters is identified using the average silhouette score. Silhouette coefficient ranges from -1 to 1, indicating whether the point is assigned to the correct cluster (positive value) or not (negative value). Equation 4 shows the calculation form of this metric using the mean intra-cluster distance (variable a) and the min nearest cluster distance (variable b) as shown in Equations 2 and 3 respectively.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

$$a(i) = \frac{1}{|c_i|-1} \sum_{\substack{j\in C_i \\ j\neq i}} d(i,j) \qquad (2)$$

$$b(i) = \min_{k\neq i} \frac{1}{|c_k|} \sum_{j\in C_k} d(i,j)) \qquad (3)$$

$$s(i) = \frac{b(i)-a(i)}{\max(a(i),b(i))} \qquad (4)$$

Figure 3 illustrates a histogram of the cohesion achieved on a set of different clusters for the *If* statements. Table 4 depicts the average number of snippets per cluster, the average cohesion of the clusters and the average number of repositories found within the cluster.

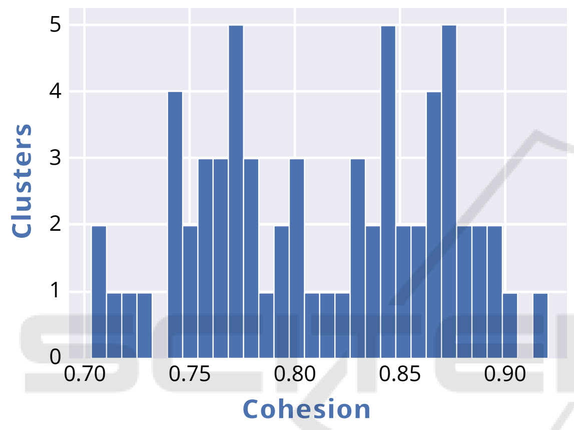

Figure 3: A histogram of the cohesion achieved in a set of different clusters.

Table 4: Clustering Results.

Type of CFS Blocks	Cluster Size	Cohesion	# Repos
If	622.3	0.81	75.8
For	202.1	0.82	69
Enhanced For	527.9	0.78	79.3
Try	299	0.86	102.5
While	195.6	0.83	100.8
Switch	93	0.78	49
Do	26	0.87	5

3.6 Cluster Selection & Generalized Form

After performing the clustering analysis, the final step of our approach examines the generated clusters and selects only those that meet the requirements of the idioms mining problem. The extracted idioms and clusters need to satisfy two specific criteria. First of all, the idioms have to be widely used by a lot of different developers and, thus, it is important to examine the number of different code repositories that contain snippets within a cluster. Additionally, the clusters derived from a clustering algorithm have to be cohesive, so that the code snippets they carry are quite similar both structurally and semantically.

Using these remarks, we define the three parameters that designate an optimal cluster:

- The number of code snippets it contains
- The number of different repositories that contain at least one snippet in the cluster
- The cluster cohesion, which is calculated as the average similarity of the snippets from the centroid. The centroid of each cluster is defined as the snippet with the lowest average distance from all the other snippets of the group. Equations 5 and 6 depict these calculations:

$$\overline{m} = \min_i\left(\frac{1}{|C|-1} \sum_{\substack{j\in C \\ j\neq i}} d(i,j)\right) \qquad (5)$$

$$cohesion = 1 - \frac{1}{|C|-1} \sum_{x\in C} d(x,\overline{m}) \qquad (6)$$

Table 5 depicts the minimum thresholds applied to each category described above. The values of these parameters were carefully selected, taking into account the number of the available snippets and the clustering results. Discarding the generated clusters that do not meet these thresholds, a total of 101 optimal clusters have been produced. From each of these optimal clusters, we extract the centroid of the cluster as the representative one, i.e. the code idiom of the respective cluster.

Table 5: Threshold Parameters per Category.

Type of CFS Blocks	# Snippets	# Repos	Cohesion
If	100	50	0.7
For	100	40	0.7
Enhanced For	100	40	0.7
Try	100	30	0.7
While	50	30	0.7
Switch	80	20	0.7
Do	25	5	0.7

The last step in our approach is to transform the generated idioms into a generalized form, that can be easily recognized and used by any developer. The main task of this step is the identification of variables, functions and objects that are commonly used within the idiom, which remain as is. On the other hand, all the other names, that change according to the domain the idiom is used on, are replaced with an abstract naming convention. For this purpose, we compute the frequency of each token of the centroid in the snippets of the corresponding cluster. All the tokens of the centroid, with a frequency lower than a specific threshold, which in our case is set to 0.5 (i.e. less than half of the snippets in the cluster contain the examined token), are replaced by an abstract token. Figure 4 depicts an example idiom extracted by our approach, as well as the generalized form it is transformed to.

```
try {
    writer . close ();
}
catch (IOException e) {
    throw new RuntimeException(e);
}
```

(a)

```
try {
    $( object ). close ();
}
catch (IOException e) {
    throw new $(method)(e);
}
```

(b)

Figure 4: (a) An example idiom extracted by our system. (b) Generalized form of the pattern adding metavariables.

Figure 5 depicts some examples of the top idioms that were extracted using our methodology, according to the number of different repositories they are found in. These abstract code snippets meet the basic requirements set for the code idioms, as they are small syntactic fragments that perform well-defined programming tasks and that are used widely by a number of different developers. Table 6 depicts the number of idioms identified in each category of the CFS.

Table 6: Number of idioms extracted per category.

Type of CFS Blocks	# Idioms
If	44
For	20
Enhanced For	15
Try	15
While	5
Switch	1
Do	1

4 EVALUATION

In this section we evaluate our methodology for extracting code idioms from the most popular GitHub repositories. The evaluation is performed in three diverse axes. At first, we examine the extracted idioms based on a set of repositories used for testing and assess their association with top repositories. Additionally, we evaluate our approach and compare our results with the respective idioms identified by Allamanis and Sutton (Allamanis and Sutton, 2014), using the *PROJECTS* dataset as baseline. Finally, towards the evaluation of the effectiveness of our approach in practice, we investigate the applicability of our extracted idioms, by employing the most popular questions and answers from *StackOverflow*.

4.1 Evaluation of Extracted Idioms

In the first step towards assessing the effectiveness of our system and the applicability of the extracted code idioms in the most popular and most (re)used repositories, we inspect the appearance of our idioms into these projects. Specifically, we examine the code blocks of the generalized code idioms against code snippets coming from a set of testing repositories, in order to identify blocks of code in these projects that are identical to the extracted idioms.

From the initial dataset coming from 1,500 of the most popular repositories, we had already left out a set of 500 repositories in the first place. We use the projects of these repositories as our main testing set, in which we examine whether they make use of our extracted idioms or not. Table 7 depicts some statistics on these repositories.

Table 7: Testing Repositories Statistics.

Metric	Value
Mean Stars	1,385.8
Mean Forks	499.2
Mean Commits	3,344.7
Mean Watches	620.7

The comparison of the code idioms identified by our approach, with code blocks from the testing repositories is performed in three steps. At first, code coming from the repositories is transformed into ASTs, which are then traversed, in order to obtain code blocks that belong to the CFSs. Next, the two ASTs, the one of the code idiom and the one of the tested code fragment, are edited and the leaves, which contain the names of the variables, methods and objects, are removed, so that we can compare only the

```
try {
    Thread.sleep($(simpleVariable1));
}
catch (InterruptedException e) {
}
```

(a)

```
if (value == null) {
    $(method1)();
}
else {
    $(method2)((String)value);
}
```

(b)

```
for (int i=0; i < $(object1).length; i++) {
    $(object1)[i]=$(method1)(i);
}
```

(c)

```
for (Thread thread : threads) {
    thread.$(method1)();
}
```

(d)

```
while ((line=$(object1).readLine()) != null){
    $(object2).$(method1)(line);
}
```

(e)

```
while ($(object1).hasNext()) {
    $(object2).add($(object1).next());
}
```

(f)

```
for (Map.Entry<String,String> entry : $(object1).entrySet()){
    $(object2).$(method1)(entry.getKey(), entry.getValue());
}
```

(g)

Figure 5: Examples of idioms extracted using our approach (a) Suspends a thread for a period of time. (b) Checks if a value is equal to null and calls the appropriate method. (c) Loops through an array and assigns its values using a method. (d) Iterates over a sequence of threads calling a method (e) Reads the content of an object line by line. (f) Adds the values of iterator to an object. (g) Iterates through the key-value pairs of a java map and calls the appropriate method of another java map object.

syntactical similarity of the two snippets. The pq-Grams algorithm is then used to calculate the distance between the two trees and, in case the resulting distance is zero, the snippets are syntactically similar and the next step can be executed. In the final step of the comparison, we examine the original variables included in the two code snippets and, specifically, we check if all the variables, excluding the ones that were replaced by meta-variables in the generalized form, of the tested snippet are also contained in the generalized idiom. In this case, the two code snippets are both structurally and semantically similar and the idiom is considered to be used by the project.

Using the aforementioned comparison approach and the set of the 500 testing repositories, it emerged that only 4 of our extracted idioms are not used at all by any repository of the testing set. In fact, each code idiom can be found at least once in 66 different repositories on average. Figure 6 illustrates the number of different code repositories where the extracted idioms are used at least once.

Additionally, in an attempt to evaluate the extent to which the extracted idioms can contribute to the software development procedure, we calculated the number of idioms identified by our methodology that are employed by the repositories of the testing set. From the results, it appears that each testing repository uses about 10 code idioms from our set at least once, while only 47 projects do not use any of our ex-

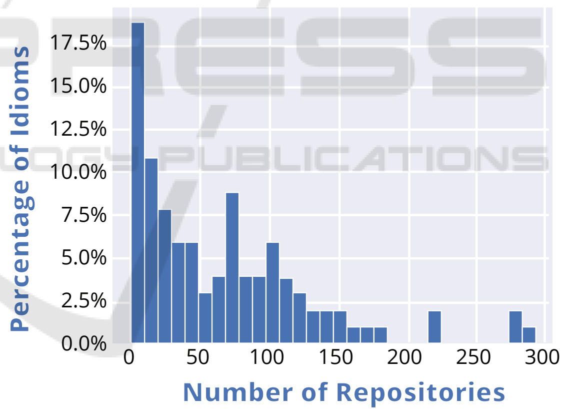

Figure 6: Number of different repositories in which our extracted idioms can be found in.

tracted idioms at all, which can be justified by the fact that the selected projects span along a wide variety of domains and functionalities. Figure 7 illustrates the number of idioms extracted by our methodology that are used at least once in the testing repositories.

4.2 Evaluation on the PROJECTS Dataset

For the evaluation of *Haggis*, their code idioms mining system, Allamanis and Sutton (Allamanis and Sutton, 2014) created two evaluation datasets, which contain some of the top open-source Java reposito-

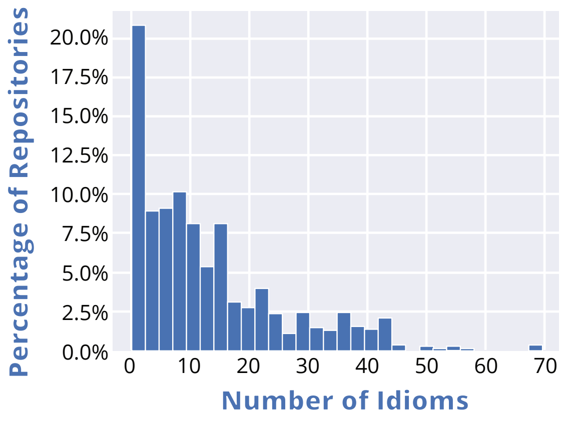

Figure 7: Number of our extracted idioms used by the repositories of the testing set.

ries from GitHub. The *LIBRARY* dataset is a selection of Java classes that make use of 15 popular Java libraries (just import the libraries without re-implementing them). The *LIBRARY* dataset was used to mine code idioms that occur across different libraries and is the main set of code blocks that was used to extract idioms. On the other hand, the *PROJECTS* dataset is a selection of the top 13 Java GitHub projects (by the time the authors accessed the code hosting platform), based on their z-score of forks and watchers. These projects were used in order to evaluate the mined idioms from the *LIBRARY* dataset. Table 8 depicts these repositories, along with the sha of the last commit that was used by the authors.

Table 8: The repositories of the *PROJECTS* dataset.

Name	Forks	Stars	Commit SHA
arduino	2,633	1,533	2757691
atmosphere	1,606	370	a0262bf
bigbluebutton	1,018	1,761	e3b6172
elasticsearch	5,972	1,534	ad547eb
grails-core	936	492	15f9114
hadoop[6]	756	742	f68ca74
hibernate	870	643	d28447e
libgdx	2,903	2,342	0c6a387
netty	2,639	1,090	3f53ba2
storm	1,534	7,928	cdb116e
vert.x	2,739	527	9f79416
voldemort	347	1,230	9ea2e95
wildfly	1,060	1,040	043d7d5

In an attempt to compare the usefulness and the applicability of the code idioms extracted by our approach against the ones identified in (Allamanis and Sutton, 2014), we evaluated our idioms also on the *PROJECTS* dataset. After cloning the project repositories and checking out in the same commit ids used in (Allamanis and Sutton, 2014), we performed the comparison analysis that was described and employed also in the previous subsection. Each code block from these projects was compared to our extracted idioms and the evaluation metric was calculated. The evaluation metric used in that case is *Precision*, defined as the percentage of the identified idioms that were found at least once in the testing repositories.

Table 9 depicts the precision of our extracted idioms against the precision achieved in (Allamanis and Sutton, 2014). Our mined idioms achieve a precision value of 58%, compared to the 50% precision value of the Haggis' idioms. Each code idiom has been found in 3.5 repositories on average.

Table 9: Evaluation results on *PROJECTS* dataset.

Approach	Precision
Haggis	50%
Ours	58%

4.3 Applicability Evaluation in Practice

Finally, in order to further assess the applicability of our methodology in practice in providing actual and useful recommendations during the development process, we made use of the *StackOverflow* questions dataset (Baltes et al., 2018). It is well-known that questions and answers in StackOverflow usually contain short code snippets that need to cover the requested functionality, so they are traditionally concise, cover only the essential code statements and represent the best programming practices. Thus, StackOverflow is considered to incorporate highly idiomatic code snippets.

From the StackOverflow questions dataset, we created two different sets of data for our evaluation. We extracted the answers referred to Java-tagged questions that contain at least one block of code and generated one set of code snippets using all code blocks found in these answers. A second set of code blocks was also build, that includes only the snippets that belong to the answers marked as correct.

We make use of the comparison method mentioned in subsection 4.1, in order to examine whether our code idioms can be found in code snippets that usually constitute idiomatic code. Finally, we use two different metrics; we calculate the *precision* of the extracted idioms, i.e. the percentage of the code idioms that are found in the StackOverflow snippets, and the

[6]The branch of the *hadoop* project that contained the selected commit id has been dropped by the developers and, thus, we did not use this project in our evaluation.

average number of times a code idiom is being used in these snippets. Table 10 depicts these results. The precision metric in the whole StackOverflow dataset is 66% and in the accepted answers 62%, while our idioms were found in 243 different posts and 30 correct answers on average. The reduced metrics on the accepted answers dataset were expected, as this set contains significantly fewer code snippets. It should be noted that Allamanis and Sutton achieved a precision of 67% on the StackOverflow answers dataset, which is almost similar to ours.

Table 10: Evaluation results on *StackOverflow* dataset.

Dataset	Precision	Average # Occurrences
Answers	66%	243
Accepted Answers	62%	30

5 THREATS TO VALIDITY

In this work, we presented our approach towards identifying and extracting maintainable and reusable code idioms from the most popular code hosting platforms. Based on the evaluation performed, the only limitation that applies to the internal validity of our system is the selection of the repositories based on their number of stars and forks, as a reflection of their reusability. At the same time, our design choice of selecting the most popular repositories from GitHub in order to mine and extract our code idioms can possibly be a threat to the external validity of our system. However, the selection of the top repositories is not random, as these projects have proven their quality in practice (including maintainability and reusability) and their popularity proves that they have the acceptance of a huge team of developers. Moreover, many researchers have argued that projects that exhibit large numbers of stars and forks, also exhibit high quality (Dimaridou et al., 2018; Dimaridou. et al., 2017; Papamichail et al., 2016b), involve frequent maintenance releases (Borges et al., 2016) and have sufficient documentation (Aggarwal et al., 2014; Weber and Luo, 2014).

As far as the external validity of our approach is concerned, one should take into account also the following limitation. We selected to look for code idioms only in CFS and not on each possible code snippet. Even though this fact can be considered as a limitation of our approach, in fact it is proven in our evaluation section that code idioms extracted from CFS are really useful.

6 CONCLUSIONS AND FUTURE WORK

In this work, we proposed a mechanism that can automatically examine the most popular projects and repositories and identify code idioms that are used within those projects. These idioms are characterized by high reusability and can be easily used by developers, in order to accelerate the software development procedure, while improving the quality of the project and ensuring an acceptable level of maintainability. As argued by Allamanis et al. (Allamanis et al., 2018), code idioms can help ”developers clearly communicate the intention of their code” and, thus, produce more readable and reusable software. Additionally, they can help developers avoid frequent errors or, even, provide useful suggestions in the most common ”how to” questions asked by developers and ”bootstrap” the quality of work of junior developers by adopting the ”best practices”.

The evaluation of our approach in three diverse axes indicates that the code idioms generated by our system can be found in a large number of the top repositories, execute useful and commonly asked programming tasks, seem natural to developers and can be actionable recommendations to the developers. Moreover, our code idioms can be found in a large degree in question-and-answers systems, such as StackOverflow, which is the main point where developers usually look for idiomatic code. Additionally, the extraction of our code idioms from the most popular repositories in GitHub ensures that these code idioms are characterized by high reusability, while, at the same time, can help the developers achieve an acceptable level of maintainability.

Future work lies in various axes. First of all, our methodology could be further expanded by including snippets of code that do no belong to CFS, in order to identify also general code idioms. Additionally, in the last step of our methodology, which produces the generalized form of the code idioms, the initialization of variables, methods and objects could also be included into the idiom. Moreover, we would suggest the use of various methods that can speed up the process of identifying code idioms, such as the use of a GPU for the distance calculations, the investigation of faster comparison methods and the additional preprocessing methods that could significantly reduce the size of the distance matrices. Finally, the usefulness of the generated idioms could be further evaluated on the basis of a developer survey.

REFERENCES

Abran, A. and Nguyenkim, H. (1993). Measurement of the maintenance process from a demand-based perspective. *J. Softw. Maintenance Res. Pract.*, 5:63–90.

Aggarwal, K., Hindle, A., and Stroulia, E. (2014). Co-evolution of project documentation and popularity within github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 360–363, New York, NY, USA. Association for Computing Machinery.

Allamanis, M., Barr, E. T., Bird, C., Devanbu, P., Marron, M., and Sutton, C. (2018). Mining semantic loop idioms. *IEEE Transactions on Software Engineering*, 44(7):651–668.

Allamanis, M. and Sutton, C. (2014). Mining idioms from source code. *CoRR*, abs/1404.0417.

Augsten, N., Böhlen, M., and Gamper, J. (2008). The ¡i¿pq¡/i¿-gram distance between ordered labeled trees. *ACM Trans. Database Syst.*, 35(1).

Augsten, N., Böhlen, M. H., and Gamper, J. (2005). Approximate matching of hierarchical data using pq-grams. In *VLDB*.

Baltes, S., Dumani, L., Treude, C., and Diehl, S. (2018). Sotorrent: reconstructing and analyzing the evolution of stack overflow posts. In Zaidman, A., Kamei, Y., and Hill, E., editors, *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 319–330. ACM.

Borges, H., Hora, A., and Valente, M. T. (2016). Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344.

Dimaridou., V., Kyprianidis., A., Papamichail., M., Diamantopoulos., T., and Symeonidis., A. (2017). Towards modeling the user-perceived quality of source code using static analysis metrics. In *Proceedings of the 12th International Conference on Software Technologies - ICSOFT,*, pages 73–84. INSTICC, SciTePress.

Dimaridou, V., Kyprianidis, A.-C., Papamichail, M., Diamantopoulos, T., and Symeonidis, A. (2018). Assessing the user-perceived quality of source code components using static analysis metrics. In Cabello, E., Cardoso, J., Maciaszek, L. A., and van Sinderen, M., editors, *Software Technologies*, pages 3–27, Cham. Springer International Publishing.

Fowkes, J. and Sutton, C. (2016). Parameter-free probabilistic api mining across github. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 254–265, New York, NY, USA. Association for Computing Machinery.

Hnatkowska, B. and Jaszczak, A. (2014). Impact of selected java idioms on source code maintainability – empirical study. In Zamojski, W., Mazurkiewicz, J., Sugier, J., Walkowiak, T., and Kacprzyk, J., editors, *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30 – July 4, 2014, Brunów, Poland*, pages 243–254, Cham. Springer International Publishing.

Ji, X., Liu, L., and Zhu, J. (2021). Code clone detection with hierarchical attentive graph embedding. *International Journal of Software Engineering and Knowledge Engineering*, 31(6):837–861. cited By 0.

Klein, P. (1998). Computing the edit-distance between unrooted ordered trees. In *ESA*.

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2:308–320.

Papamichail, M., Diamantopoulos, T., and Symeonidis, A. (2016a). User-perceived source code quality estimation based on static analysis metrics. pages 100–107.

Papamichail, M., Diamantopoulos, T., and Symeonidis, A. (2016b). User-perceived source code quality estimation based on static analysis metrics. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 100–107.

Sivaraman, A., Abreu, R., Scott, A., Akomolede, T., and Chandra, S. (2021). Mining idioms in the wild. *CoRR*, abs/2107.06402.

Tai, K. (1979). The tree-to-tree correction problem. *J. ACM*, 26:422–433.

Tanaka, H., Matsumoto, S., and Kusumoto, S. (2019). A study on the current status of functional idioms in java. *IEICE Transactions on Information and Systems*, E102.D:2414–2422.

Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., and Zhang, D. (2013). Mining succinct and high-coverage api usage patterns from source code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 319–328.

Weber, S. and Luo, J. (2014). What makes an open source code popular on git hub? In *2014 IEEE International Conference on Data Mining Workshop*, pages 851–855.

Zhang, K. and Shasha, D. (1989). Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18:1245–1262.

Zhang, Y. and Wang, T. (2021). Cceyes: An effective tool for code clone detection on large-scale open source repositories. pages 61–70. cited By 0.