DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Recognition of Generated Code in Open Source Software

Marcel Bruckner

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Recognition of Generated Code in Open Source Software

Erkennung von generiertem Code in Open-Source Software

Author:	Marcel Bruckner
Supervisor:	Broy, Manfred; Prof. Dr. rer. nat. habil.
Advisor:	Jürgens, Elmar; Dr. rer. nat.
Submission Date:	15.10.2018

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15.10.2018 Marcel Bruckner

Acknowledgments

My thanks go to my advisor Dr. Elmar Jürgens and my mentor Dr. Alexander Rhein for always supporting me at all phases of this thesis. Additionally do I thank the CQSE GmbH for giving me the opportunity to write my thesis in an environment where I could bring in my own ideas and could always rely on a highly competent team. Further I thank Prof. Dr. Manfred Broy for supervising me on this fascinating topic.

Abstract

A suffix-tree clone-detection approach to find clones among the comments in source code files is presented. Using this approach we found a multitude of generator-patterns that are inserted by their respective code generator and identify the source code file as generated. We extracted the patterns to build up a generator-pattern repository that can be used to automatically classify code into the categories *generated* and *manually maintained*. This repository is used on a reference data set and a huge, randomly composed collection of open source projects to test its capabilities and to calculate different proportions of generated code.

Contents

1 Introduction

Source code of software can be categorized with respect to its role in the maintainability of a software system. The biggest categories besides manually produced production code, which is written and maintained by hand of a developer, are generated code and test-code. In many systems up to 50% of the source code arise in these categories.

Static analysis detects problems in the quality of code. At the same time the code category is essential for the relevance of the examined quality criterion. Security flaws and performance problems which are crucial in production code can be irrelevant in generated code since it is not directly edited during maintenance. To enhance the relevance and significance of the results of static analysis the category of the examined source code has to be taken into account.

This applies particularly in benchmarks that investigate the frequency of the occurrence of quality defects and the distribution of metric values in a variety of software projects. Since a manual classification of source code in a multitude of projects is not feasible in practice an automated approach is necessary.

This work targets the conception and prototypical implementation of an automatic detection of generated code which includes the following steps:

- A prototypical implementation of heuristics to detect generated code which use techniques of clone-detection on the comments that are extracted from the source code.
- A list of patterns that detect generated code of several code generators will be derived by means of the comments. To this end a generator pattern repository will be created.
- The completeness and accuracy of the developed patterns will be evaluated on a reference data set.
  This data set is "a large curated collection of open source Java systems. The corpus reduces the cost of performing large empirical studies of code and supports comparison of measurements of the same artifacts."[1, p. 1]. It contains a total of 112 projects maintained by a multitude of big software companies. These projects are written in Java and have a total of 33.971.977 lines of code ranging form 6.991 up to 7.142.778.

- Automatic classification of source code in a variety of open source systems and evaluation of the amount of generated code.
  We gathered a huge, randomly composed collection of open source projects containing a total of 1.112 projects written in 18 different programming languages. The projects contain a total of 23.638.640 lines of code, whereas they are distributed among the programming languages ranging from 2.738 up to 6.828.723 lines of code.

2 Terms and Definitions

This chapter is to explain important terms and definitions used in the following chapters. It clears up the terminology and metrics used and gives an overview over Teamscale.

2.1 Terminology

We define the used terms and definitions to avoid the reader to be confused by arising terminology and to clarify the concrete definition we refer to when using specific terms.

2.1.1 Generated code

As proposed in [2] "we consider *generated code* all artifacts that are automatically created and modified by a tool using other artifacts as input. Common examples are parsers (generated from grammars), data access layers (generated from different models such as UML, database schemas or web-service specifications), mock objects or test code."[2, p. 11]

2.1.2 Manually-maintained code

In contrast we consider *manually-maintained code* as suggested in [2] "all artifacts that have been created or modified by a developer either under development or during maintenance. This includes all artifacts created using any type of tool (e.g. editor). Configuration, experimental or temporary artifacts, if created or modified by a developer, should also be considered as *manually-maintained code*."[2, p. 11]

2.1.3 Generator pattern

As a *generator pattern* we consider all comments that a code generator adds to the generated artifacts during generation and that distinguish generated from manually-maintained code. Most code generators do add comments that are characteristic for the generator and identify the code as generated. This includes header comments preceding entire source code files as well as comments that mark single functions or datatype definitions.

2.1.4 Generator pattern repository

The *generator pattern repository* is one aim of this thesis. It is a database holding the generator patterns associated to its generator. It can be used to detect generated source code files efficient and reliable.

2.1.5 Clone

We refer to *clones* in the context of this thesis as text fragments that have two or more instances in comments. This includes whole comments that are identical in different source code files as well as parts of comments. The original and the duplicated fragment build a clone pair [3].

2.1.6 Clone chunk

For the use in Esko Ukkonens algorithm for the construction of suffix-trees [4] the text of the comments has to be normalized in *clone chunks*. The suffix-tree clone-detection approach uses a sequence of clone chunks and constructs the tree on them. To be suitable each comment is split in separate words, whereas additional information gets appended to each word that will later on be used to identify the location of the word in the original comment. This information includes the uniform path to the original source code file, the line number in which the word originated and the programming language.

2.1.7 Sentinel

A boolean comparison for equality of a sentinel and another object will always evaluate to false. It is used on the one site to separate coherent sequences of clone chunks from each other, and on the other hand to convert an implicit suffix-tree into an explicit one. A *sentinel* is a subclass of the *clone chunk* class.

2.1.8 Clone class

A *clone class* is the maximal set of comments in which any two of the comments form a *clone pair*. Table 2.1 depicts an example of the appearance of 3 clone classes.

2.2 Metrics

The *Lines of Code (LOC)* metric represents the size of a source code file so that it can be compared to other files easily. We use it to calculate the overall sizes of the different

I dont know how to make code snippets out of it. Tried many ways but listings in tables is somewhat awful.

Table 2.1: All same colored cells form a clone class. Any two instances of a clone in a clone class form a clone pair.

Fragment 1	Fragment 2	Fragment 3
/* The following code was generated by . . . */	/* The following code was generated by . . . */	/** This character denotes the end of file */
/* The content of this method is always regenerated */	/* The content of this method is always regenerated */	/* The content of this method is always regenerated */
/** Translates characters to character classes */	/** This class is a scanner generated by . . . */	/** This class is a scanner generated by . . . */

code categories to compare the proportions they yield in total. Additionally we use it to calculate the average file size of generated and manually maintained source code.

2.3 Teamscale

Teamscale is developed by the CQSE GmbH which was founded in 2009 as spin-off of Technical University Munich (TUM). They offer innovative consulting and products to help their customers evaluate, control and improve their software quality.
Their main product is Teamscale which is a tool to analyze the quality of code with a variety of static code analyses. It helps to monitor the quality of code over time and is personally configurable to allow users to focus on personal quality goals and keep an eye on the current quality trend.
The supported programming languages of Teamscale are *ABAP, Groovy, Matlab, Simulink/StateFlow, Ada, Gosu, Open CL, SQLScript, C#, IEC 61131-3 ST, OScript, Swift, C/C++, Java, PHP, TypeScript, Cobol, JavaScript, PL/SQL, Visual Basic .NET, Delphi, Kotlin, Python, Xtend, Fortran, Magik* and *Rust*.
One major aspect in the quality analysis performed by Teamscale is the *clone-detection*. With it duplicated code created by copy & paste can be found automatically.

Table 2.3: Token properties

Property	Description
Text	The original input text of the token, copied verbatim from the source.
Offset	The number of characters before this token in the text. The offset is 0-based and inclusive.
End Offset	The number of characters before the end of this token in the text (i.e. the 0-based index of the last character, inclusive).
Origin ID	The string that identifies the origin of this token. This can, e.g., be a uniform path to the resource. Its actual content depends on how the token gets constructed.
Type	The type of the token.
Language	The programming language in which the source code file is written that contains the token.

2.3.1 Suffix-tree clone-detection

In [4], an on-line algorithm is presented for constructing the suffix-tree for a given string in time linear in the length of the string. Based on this data structure a string-matching algorithm is presented in [5]. These two algorithms have been extended to be usable in this thesis to detect clones among comments in source code.

2.3.2 Lexer

A tokenizer, also called lexical scanner (short: *Lexer*), is a programm that splits plain text in a sequence of logical concatenated units, so called tokens. The plain text tokenized by the lexer can be anything, but we will restrict it to source code. Teamscale includes a variety of lexers for every supported programming language.

2.3.3 Token

Tokens in the context of Teamscale are objects that are returned as a sequence by the lexer. They provide a data structure holding the main properties of the smallest possible units a source code file can be split into. An overview of these properties is shown in Table 2.3.

2.3.4 Token class

A token is always an element of one of the token classes *LITERAL*, *KEYWORD*, *IDENTIFIER*, *DELIMITER*, *COMMENT*, *SPECIAL*, *ERROR*, *WHITESPACE* or *SYNTHETIC*. These are mutually exclusive sets that wrap all possible types a token can adopt.

2.3.5 Token type

A token always adopts one single type. These range from the simple *INTEGER_LITERAL* over *HEADER* to *DOCUMENTATION_COMMENT*.

3 Related Work

Our work is done in a field where much research is performed. This makes our work easily comparable to other ones, especially the one provided in the paper presented in the following subsections.

3.1 Automatic Categorization of Source Code in Open-Source Software - Jonathan Bernwieser

In his thesis, Johnathan Bernwieser introduced an attempt of automating the process of categorization of source code in [6]. He classified source code in *productive* and *test* code, followed by a sub-categorization in *manually maintained* and *automatically generated*.
He ran in several problems during his research, especially with the identification of generated code. Different to test code, which often follows the naming convention to include the word *test* into the class name or that the test classes do follow inheritance lines which identify them, generated code has no standardized way of being marked by the respective code generator.
In his approach he used a filtering of the classes which used the observation that code generators often include the term *generated by* in their comments. It quickly turned out that this approach generated many false positives as software developers often also use this term in their documentation. Nevertheless he pointed out that further investigation on the comments, especially in the ones including this term, might result in another way to find generated code. This thesis deploys his finding.
Due to the high number of false-positive categorizations of generated code resulting from this simple assumption, Bernwieser tackled the problem by using clone-detection on source code. He examined the question if a high clone coverage respectively higher clone density implies generated classes.
Therefore he implemented two mechanisms that examined source code for the following clone metrics.

3.1.1 Number of clones per clone class

This metric gave very interesting results on the data set used by Bernwieser. It turned out that generated code often has many instances among projects due to the usage of

templates and routines from which it gets generated and what makes the resulting source code identically between generations. The problem that he ran into was the fact that there exists also generated code with only a small number of instances which resulted in a minimum filter threshold that had to be really low to detect all generated classes, which made it impractical to use.

3.1.2 Clone coverage per class

In this approach Bernwieser tried different clone coverage thresholds and lines of code limits to detect generated code. But the resulting problem is that there exists no exact correlation between the size of a source code file and the possibility of it containing generated code. Furthermore it showed that the size of a source code file is irrelevant for being a generated class. This approach produced a high number of false negatives and thus filtered out generated code which should have been detected.

3.1.3 Results

Bernwieser showed that there is indeed a correlation between clone coverage and code generation, but at this point there was no way found on how to use this correlation to automate code generator identification. Anyhow he pointed out the fact that many code generators add specific comments to the generated files he found and on which he depicted further research could be done.

3.1.4 Differences in the approaches

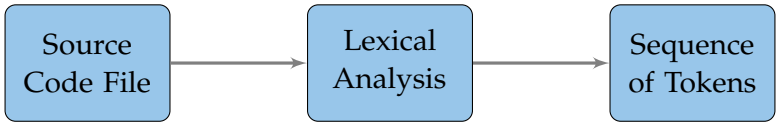
In this thesis, the approach differs from Bernwiesers so that the clone-detection is done on the comments added to the source code files in the data set, rather than on the respective source code. A goal of our approach is to find the specific generator patterns that Bernwieser pointed at. It's based on his observation that the same code generators do always add the same characteristic generator patterns regardless of what template or routine the generator used for generation. Resulting on this the target of this thesis is to find these patterns by using clone-detection on comments. The expected result is that the clone classes with the highest number of instances will be the specific generator patterns.

4 Approach

One aim of this thesis is the prototypical implementation of heuristics to detect generated code which uses techniques of clone-detection on the comments that are extracted from the source code. A list of patterns that detect generated code of several code generators is derived based on comments in source code and we created a generator-pattern repository. This repository provides a database of code generators and their respective characteristic generator-pattern which identifies the the generated source code.
We use a Teamscale server to perform the basic analysis tasks on the specified projects.

4.1 Use Teamscale as Lexer to extract Tokens

The first step in the approach used in this paper is the lexical analysis of the source code included in the projects. Teamscale comes with a multitude of lexers applicable for many different programming languages. The general workflow in this step is reading each source code file found in the project, perform the lexical analysis and tokenize the source code file into a sequence of logically coherent tokens representing the source code on a logical level. The tokens are saved server-side in the Teamscale instance.



4.2 Connect to the Teamscale server and retrieve comments

In the second step the tool written for this thesis connects to the Teamscale instance and retrieves the comments.
This has to be done in the following steps:

1. Get the projects that are currently available in the Teamscale instance.

2. For each project retrieve the respective uniform paths[1] to access the source code files on the server.
3. For each uniform path retrieve the comments that are included in the respective source code file. In this step the server filters all tokens that are available for each file. The only important token class is the *COMMENT* class, which itself contains the sub-classes shown in Table 4.1.
4. The local file path of every source code file is retrieved from the server based on the uniform path. This will get important later in the evaluation.
5. Each local file path gets associated to the respective `List` of comments.

Retrieving the comments is highly multithreaded to speed it up. For benchmarks see 5.4.3.

Table 4.1: Comment types

Token type	**Example**
HASH_COMMENT	*# Sample PHP script accessing HyperSQL through the ODBC extension module.*
DOCUMENTATION_COMMENT	*/** Generated By:JJTree: Do not edit this line. */*
SHEBANG_LINE	*#!/usr/bin/env python*
TRIPLE_SLASH_DIRECTIVE	*/// <reference path="Parser.ts" />*
TRADITIONAL_COMMENT	*/* Do not modify this code */*
MULTILINE_COMMENT	*"""Testsuite for TokenRewriteStream class."""*
END_OF_LINE_COMMENT	*// don't care about docstrings*
SIX_COLUMNS_COMMENT	**Ich verstehe leider nicht wann dieser Typ verwendet wird**

[1] A uniform path is the path the Teamscale instance uses to represent the source code file. The URL to access the file on the server can be build using it.

4.3 Filter generated files

With every iteration of the algorithm the generator-pattern repository grows and more patterns are found. These patterns are used to filter the source code files and remove the already detected generated files. This reduces the amount of work that has to be done by the following steps.
Additionally, the amount of found presumable generator-patterns is reduced drastically when we sort out the generated files before applying the follwing steps.

4.4 Prepare comments for suffix-tree clone-detection

Once the comments are received from the Teamscale server instance they have to get prepared to be usable in the suffix-tree clone-detection as introduced by Esko Ukkonen [4][5].
The preparation is highly multithreaded to speed it up. For benchmarks see 5.4.3.

4.4.1 Create thread for each source code file

The step described in Section 4.2 produces a `Map` in which every local path to a source code file is associated with its respective `List` of comments. In the first part of the preparation one thread is generated for each source code file respective local file path. These threads will be responsible for the preparation of each single file and the results will be merged at the end.

4.4.2 Convert comments into CloneChunks

At first the comments are split at the linebreaks into single lines and leading and trailing whitespaces of each line are removed. The resulting lines are in turn split at the whitespaces separating single words and expressions.
The `List` of words gets now iterated and every word gets checked if it holds any valuable information. To evaluate the value of each word it is checked if it contains alphabetic letters *(a-z, A-Z)* or digits. If a word only consists of nonalphabetic characters and no digits it gets sorted out *(e.g. "--------", "/**", "<!–")*.
The remaining words that bring a value to the approach are now converted into `CloneChunk`s. Therefore is added to every word the type of the comment, the offset and the line number of the comment and the local file path the comment originated from.
For an overview over these properties of a comment see 2.3.

4.4.3 Add sentinels

To prevent the suffix-tree clone-detection algorithm from intermixing the comments a sentinel [2.1.7] gets added at the end of the `List` of `CloneChunk`s resulting from the previous steps. This is necessary due to the fact that the algorithm works an a single `List` of `CloneChunk`s that is build from merging all `CloneChunk`s from every comment.

4.4.4 Merge thread results

Once all threads are finished, resulting in all comments being converted into `List`s of `CloneChunk`s and sentinels being added to the end of each `List`, all `List`s get concatenated into one huge `List` containing all words of all comments from all projects. On this `List` the suffix-tree clone-detection algorithm can now be applied.

4.5 Build suffix-tree

Esko Ukkonen introduced an algorithm to construct suffix-trees on strings in his paper for *Algorithmica* in 1995 [4]. "A suffix-tree is a trie-like structure representing all suffixes of a string. [...] The new algorithm [...] processes the string symbol by symbol from left to right [...]. The algorithm is based on the simple observation that the suffixes of a string $T^i = t_1 \dots t_i$ can be obtained from the suffixes of string $T^{i-1} = t_1 \dots t_{i-1}$ by catenating symbol t_i at the end of each suffix of T^{i-1} and by adding the empty suffix." [4, p. 2]
For this thesis the algorithm has been expanded to work on `List`s of arbitrary data types. The simple observation that lead to the expansion of the algorithm is that a string is just a `List` of characters. When building the suffix-tree this `List` gets traversed from left to right and each `CloneChunk` is added and the suffix-tree is expanded.
We used this property of the algorithm to extend it to a generic version of it by allowing `List`s of any data types. The only mandatory property the data type must provide is an equality function that can be evaluated in linear time to keep the linear time property of the algorithm. We implemented a function that calculates a hash value of the data structure and performs a simple *integer - integer* comparison that keeps the linear time property.
Figure 4.1 illustrates the implicit suffix-tree on the sequence of `CloneChunk`s representing the `List` of words *Do not modify ... Do not modify*. After adding a sentinel at the end of the `List` the suffix-tree becomes explicit, which is used in this thesis and shown in Figure 4.2.

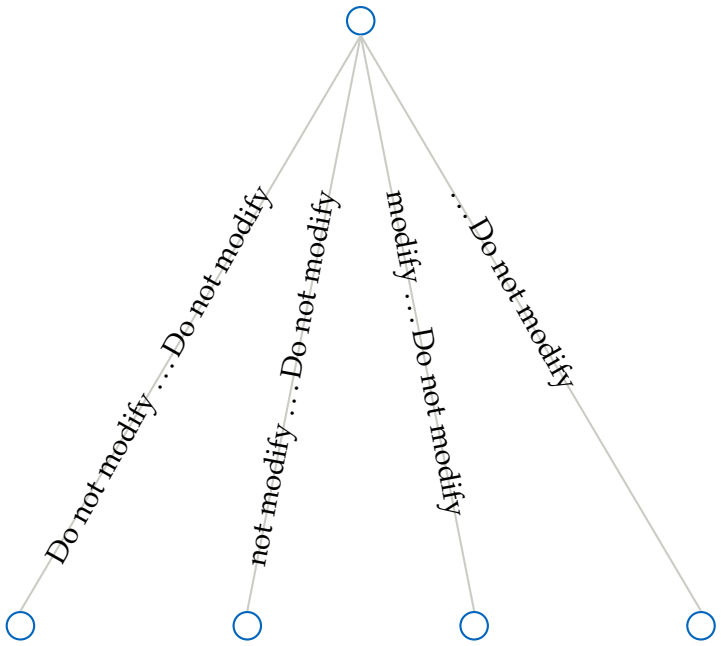


Figure 4.1: The implicit suffix tree for the list of words "Do not modify ... Do not modify".

For a deeper insight into the details of the algorithm for suffix-tree construction see [4].

4.6 Find clones

To find duplicate sequences in the suffix-tree, a variety of tree search algorithms can be applied. They are all based on the observation that the clones are represented by the paths in the suffix-tree containing the searched sequence. Listing 4.1 can be used to perform a breadth first search to find clones.
When the graph search is performed the `CloneClass`es get returned as a `List` whereas each entry represents a `CloneClass`. Each `CloneClass` itself is a `List` of `List` of `CloneChunk`s. Each inner `List` of the `CloneClass` represents a clone instance. The clones consist of a `List` of `CloneChunk`s which have the additional information added to the string representation as shown in Listing 4.2. So every clone that is represented by the `List` named *members* can be associated to the file it originated from. This will become important in the next steps.
This `List` of `CloneChunk`s gets now converted into a `List` of `CloneResult`.

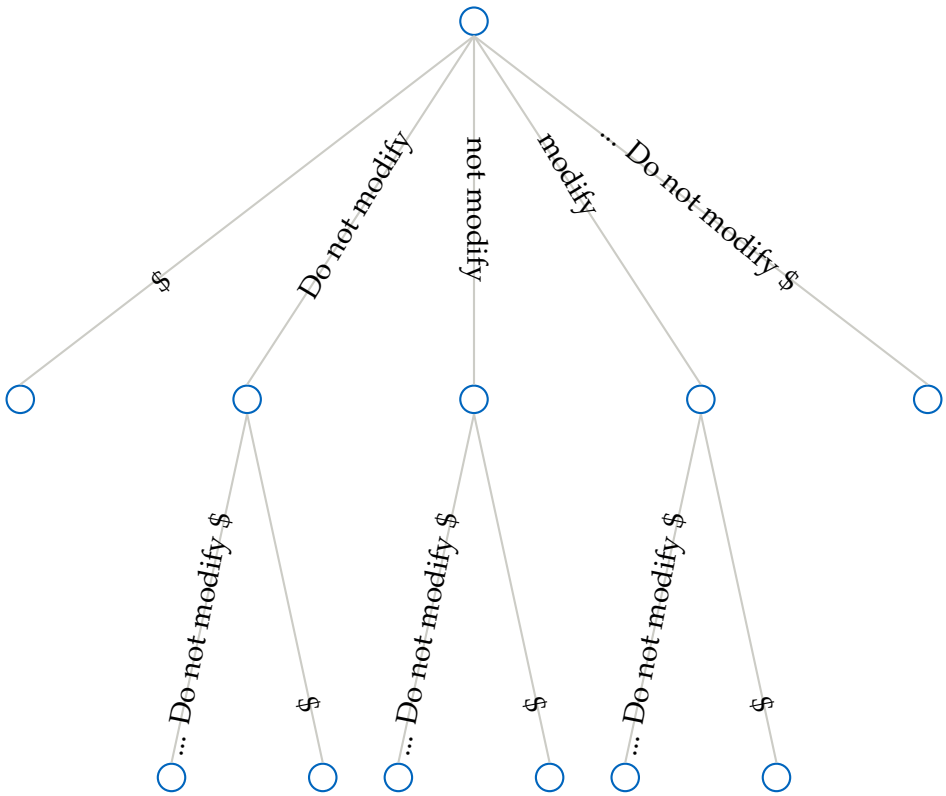


Figure 4.2: The explicit suffix tree for the list of words "Do not modify ... Do not modify$". The $ symbol represents the added sentinel used to make the tree explicit.

A `CloneResult` is a more compact and convenient representation of the `CloneClass`es. Each has the text it represents as its *name* and a `List` of all occurrences associated, whereas a occurrence is a `Pair` representing the *originID* (file identifier) and the *line number*.
The conversion is highly multithreaded to speed it up. For benchmarks see 5.4.3.

Listing 4.1: Recursive Breadth First search algorithm to find all containing branches in a suffix tree for a given sequence.

```
for each (childNode in currentNode.children)
        if (childNode.label does not match sequence)
                continue
        if (childNode.label fully matches sequence)
                return all childBranches
        if (childNode.label partially matches sequence)
                recurse on childNode
return no match
```

Listing 4.2: The `CloneClass`es as returned by the graph search algorithm.

```
{
  "cloneClasses": [
    "members": [
      ["Do", "not", "modify"],
      ["Do", "not", "modify"],
      ["Do", "not", "modify"]],
    "members": [
      ["This", "class", "is", "generated", "by"],
      ["This", "class", "is", "generated", "by"],
      ["This", "class", "is", "generated", "by"]],
    "members": [
      ["The", "following", "code", "was", "generated", "by"],
      ["The", "following", "code", "was", "generated", "by"],
      ["The", "following", "code", "was", "generated", "by"]],
   ]
}
```

Listing 4.3: CloneResults after processing of the CloneClasses.

```
{
  "name": "Do not modify", "occurrences": [
    [originID: "/projectX/class1.java", lineNumber: "9"]
    [originID: "/projectX/class2.java", lineNumber: "9"]
    [originID: "/projectX/class3.java", lineNumber: "9"]],
  "name": "This class is generated by", "occurrences": [
    [originID: "/projectY/class1.java", lineNumber: "1"]
    [originID: "/projectY/class2.java", lineNumber: "1"]
    [originID: "/projectY/class3.java", lineNumber: "1"]],
  "name": "The following code was generated by", "occurrences": [
    [originID: "/projectZ/class1.java", lineNumber: "48"]
    [originID: "/projectZ/class2.java", lineNumber: "48"]
    [originID: "/projectZ/class3.java", lineNumber: "48"]],
}
```

4.7 Reduce amount of not valuable data

The raw suffix-tree clone-detection produces a lot of false positive CloneResults. This is due to the fact that the clone-detection cannot distinguish between common phrases and actual generator-patterns. To reduce the amount of not valuable and redundant data three steps are performed.

4.7.1 Filter possibly generated CloneResults

In this steps the CloneResults get filtered so that possibly generated ones can easily be distinguished from the ones that are unlikely to be generated. This step is only to improve and speed up the manual search for generator-patterns in the found clones performed later.
As stated in [6] most generator-patterns will include the word *generated*. From this observation a filter pattern has been derived and extended which is shown in Listing 4.4. With this pattern the List of CloneResult gets filtered by their names.

Listing 4.4: Pattern to filter possibly generated CloneClasses.

```
"(Do not (modify|edit|change))|(generate(d)?)"
```

4.7.2 Accumulation of CloneResults

Due to the fact that the clone-detection on suffix-trees can only settle on a minimum clone length, but not on a maximum length, an accumulation is done.
As seen in Listing 4.5 the *name* of the first `CloneResult` is a substring of the *names* of the other two. We observed that the sets of occurrences of the `CloneResult`s do overlap in all cases, which is illustrated in Listing 4.5. So to speed up and remove redundant data only the `CloneResult` with the shortest name length is kept and the sets of occurrences from all `CloneResult`s that include the name as a substring get merged into that `CloneResult`.
As illustrated in Listing 4.6, the amount of data stored is drastically reduced what speeds up processing later on.

Listing 4.5: The `CloneResult`s before accumulation with a minimum clone length of 2.

```
1  {
2          "name": "generated by", "occurrences": [
3                  {originID: "/projectX/class1.java", lineNumber: "9"}
4                  {originID: "/projectX/class2.java", lineNumber: "9"}
5                  {originID: "/projectX/class3.java", lineNumber: "9"}],
6          "name": "is generated by", "occurrences": [
7                  {originID: "/projectX/class1.java", lineNumber: "9"}
8                  {originID: "/projectX/class2.java", lineNumber: "9"}
9                  {originID: "/projectX/class3.java", lineNumber: "9"}],
10         "name": "This class is generated by", "occurrences": [
11                 {originID: "/projectX/class1.java", lineNumber: "9"}
12                 {originID: "/projectX/class2.java", lineNumber: "9"}
13                 {originID: "/projectX/class3.java", lineNumber: "9"}],
14 }
```

Listing 4.6: The `CloneResult`s after accumulation with a minimum clone length of 2.

```
1  {
2          "name": "generated by", "occurrences": [
3                  {originID: "/projectX/class1.java", lineNumber: "9"}
4                  {originID: "/projectX/class2.java", lineNumber: "9"}
5                  {originID: "/projectX/class3.java", lineNumber: "9"}]
6  }
```

4.7.3 Clustering of CloneResults

During the search for generator-patterns we realized that the generated files always have the same structure, resulting from the fact that generators mostly reuse the same templates that are only filled with user-dependent information.
This results in the observation that most `CloneResult`s will have their occurrences always at the same line in the source code file, only having a small derivation coming from additional license headers or package declarations.
To check whether the `CloneResult` is likely to be a generator-pattern we iterate over all occurrences to sort out the ones that only occur once in their respective line. Furthermore we added a threshold for the maximal distance between the occurrences of ten lines, what sorts out most of the common phrases and keeps all generator-patterns.

4.8 Generate links to the files

The last step performed by the tool is to save the `CloneResult`s to be further processed. This gets done in three separate steps:

1. Save as repository

   A `Java` class is generated that holds a `List` of `Strings` in which a human readable version of the presumable generator-pattern is saved together with a escaped version that can be used in regex search for later usage as the generator-pattern repository.

2. Save as files

   The exact folder structure of the original projects is saved, but removing all files that do not contain the presumable generator-patterns. In these newly created projects one can easily review the found occurrences and check for the algorithms reliability. Additionally, a plain `XML` list of all presumable generated files is saved.

3. Save as links

   The `CloneResult`s are sorted by their number of occurrences and for each number a folder is created. Within these folders, we created subfolders labeled by the name of the included possible generator-pattern. In these subfolders, links to all files containing the generator-pattern are created.

Save as files and as links is highly multithreaded to speed it up. For benchmarks see 5.4.3.

4.9 Creation of a Generator-Pattern Repository

When the tool is finished and the raw data is saved it needs to be reviewed by hand.
We searched the folders including the links to the generator-patterns (sorted by their number of occurrences) in descending order due to the fact that a high number of occurrences implies many repetitions of the comment, which is a possible sign of it being generated.
By looking at the comments including the found presumable generator-patterns we found a variety of different generator-patterns. Table 4.3 shows an excerpt of the actual generator-pattern repository associated to their respective generator.
As stated in [6], the word *generate* used to filter the `CloneResult`s is not a sufficient criterion to decide whether a source code file is generated. Due to this fact the algorithm produces many false positives and patterns that use the word *generated*, but in the context that it describes that the following function or class generates something, rather than being generated itself.
We reviewed the presumable generator-patterns and quickly found multiple comments that mark the source code file as generated. These were collected in the Generator-Pattern Repository.
Based on the repository we could iteratively find more patterns. To do so we repeated all steps described in the approach, but checked the source code files directly after retrieving them from the Teamscale instance. The source code files containing one or multiple of the generator-patterns were sorted out, so the suffix-tree clone-detection has been performed on a shrinking set of source code files containing less generated code in every iteration.
We repeated searching for generator-patterns and adding them to the repository and the suffix-tree clone-detection until no generator-patterns were left.

Table 4.3: An excerpt of the Generator-Pattern Repository

Generator	Regular Expression Pattern
Apache Axis	This file was auto-generated from WSDL by the Apache Axis WSDL2Java emitter.
Apache Cayenne	It is (probably)?a good idea to avoid changing this class manually(,)? since it (((may)|(will)))be overwritten next time code is regenerated. If you need to make any customizations(,)? (((please use)|(put them in a)))subclass.
Apache Thrift	Autogenerated by Thrift DO NOT EDIT UNLESS YOU ARE SURE THAT YOU KNOW WHAT YOU ARE DOING
CUP	The following code was generated by CUP <Version><Timestamp>
Eclipse	TODO Auto-generated method stub
Java Annotation	@generated(modifiable)?
JavaCC	This class is generated by JavaCC.
JavaNCSS	WARNING TO <Project>DEVELOPERS DO NOT MODIFY THIS FILE! MODIFY THE FILES UNDER THE JAVANCSS DIRECTORY LOCATED AT THE ROOT OF THE <Project>PROJECT. FOLLOW THE PROCEDURE FOR MERGING THE LATEST JAVANCSS INTO <Project>LOCATED AT javancss/coberturaREADME.txt
JAXB	This file was generated by the JavaTM Architecture for XML Binding$JAXB$ Reference Implementation,(<Version>See http://java.sun.com/xml/jaxb Any modifications to this file will be lost upon recompilation of the source schema. Generated on: <Timestamp>)?
JFlex	NOTE: This class was automatically generated. DO NOT MODIFY.
NetBeans	Do NOT modify this code. The content of this method is always regenerated by the Form Editor.
SableCC	This file was generated by SableCC ($http://www.sablecc.org/$.)?
schemagen	@author Auto-generated by schemagen on <Timestamp>
Snowball	This file was generated automatically by the Snowball to Java compiler

5 Evaluation

5.1 Tasks to evaluate

5.1.1 Evaluation of the completeness and accuracy of the produced heuristics against a reference data set

To evaluate the completeness and accuracy of the heuristics produced during the development of this thesis a reference data set has been used.
We implemented a procedure to check all files in the projects whether it contains a generator-pattern. These files are separately saved in a list. Bernwieser [6] provided a list of files that he detected as generated to us, to witch our results are compared against.
We do this to evaluate whether the approach given in this thesis is an improvement to the existing ones, especially the one given by [6]. It resolves whether the generator-pattern repository can be used as a database in production while performing static code analysis.

5.1.2 Automatic classification of source code in a huge collection of open source systems & evaluation of the ratio of generated code

To test the generator patterns stored in the repository a variety of open source projects have been acquired.
These projects have been filtered using the generator-pattern repository to detect the amount of generated code in a huge, randomly composed collection of open source systems.
We calculated the amount of generated code in these projects to evaluate the usage of code generators.

5.2 Study Objects

We used different study objects to test the approach used in this thesis and to evaluate the completeness and accuracy of the generator-pattern repository.
A reference data set is described in Section 5.2.1 which includes good documented

code, whereas Section 5.2.2 describes a huge, randomly composed collection of open source projects that is used to test the generator-pattern repository in a particularly not curated environment.

5.2.1 Qualitas Corpus

The Qualitas Corpus is "a large curated collection of open source Java systems. The corpus reduces the cost of performing large empirical studies of code and supports comparison of measurements of the same artifacts."[1, p. 1]
The current release is from the year 2013 and includes 112 different projects. To be easily comparable to the work in [6], we also added the project Mahout [7].
Additionally, we could perform the analysis on all projects so we didn't have to exclude *eclipse_SDK* nor *netbeans*.
Table 5.1 displays all projects that are included in the Qualitas Corpus together with their amount of lines of code. The size of the projects ranges from 6.991 *(fitjava)* up to 7.142.778 *(netbeans)*.

5.2.2 Random Projects

To collect a huge, randomly composed collection of open source projects a git crawler has been implemented. By passing a keyword to the crawler it queries GitHub to return project descriptions of projects that contain the keyword in their names or descriptions. From the received project descriptions the links to clone the projects are extracted. The projects are cloned and saved sorted by their programming languages, whereas the crawler queries for projects written in all languages supported by Teamscale to provide the highest possible level of randomness and diversity for the projects.
Table 5.3 displays an overview over the projects used.

5.3 Study Design

To evaluate the completeness and accuracy of the produced heuristics we used the Qualitas Corpus to calculate the amount of generated code. Using this information we compared our results to the ones provided in [6] to derive the improvement our approach makes in distinguishing generated from manually maintained code.
Based on the set of generated source code files we calculated different metrics:

- Lines of code for generated and manually maintained source code.
- Number of generated and manually maintained source code files.
- Ratio of lines of code per source code files in the categories.

Table 5.1: All projects of the Qualitas Corpus and their respective lines of code.

Project	LOC	Project	LOC	Project	LOC
ant	256.041	antlr	58.935	aoi	153.186
argouml	389.952	aspectj	598.485	axion	41.862
azureus	831.586	batik	366.507	c_jdbc	174.972
castor	349.301	cayenne	341.902	checkstyle	89.922
cobertura	68.154	collections	109.415	colt	84.592
columba	149.498	compiere	727.702	derby	1.138.858
displaytag	38.729	drawswf	46.540	drjava	160.308
eclipse_SDK	4.956.920	emma	39.676	exoportal	146.947
findbugs	185.806	fitjava	6.991	fitlibraryforfitnes	42.233
freecol	205.085	freecs	29.943	freemind	86.244
galleon	135.442	ganttproject	69.322	gt2	1.514.789
hadoop	444.593	heritrix	126.652	hibernate	711.370
hsqldb	269.978	htmlunit	174.415	informa	29.587
ireport	338.819	itext	145.118	ivatagroupware	71.851
jag	28.957	james	83.716	jasml	7.272
jasperreports	347.502	javacc	35.145	jboss	968.808
jchempaint	372.743	jedit	176.672	jena	164.679
jext	100.210	jFin_DateMath	16.686	jfreechart	313.268
jgraph	59.145	jgraphpad	43.652	jgrapht	41.887
jgroups	137.614	jhotdraw	133.830	jmeter	182.552
jmoney	11.304	joggplayer	51.654	jparse	32.270
jpf	22.521	jrat	31.084	jrefactory	301.940
jruby	244.774	jspwiki	110.005	jstock	74.361
jsXe	35.307	jtopen	645.715	jung	67.024
junit	13.359	log4j	68.612	lucene	611.422
mahout	177.325	marauroa	36.859	maven	111.581
megamek	336.267	mvnforum	172.855	myfaces_core	189.954
nakedobjects	214.777	nekohtml	13.342	netbeans	7.142.778
openjms	111.837	oscache	19.702	picocontainer	15.999
pmd	80.971	poi	363.487	pooka	72.167
proguard	101.330	quartz	62.229	quickserver	30.239
quilt	13.035	roller	135.210	rssowl	174.209
sablecc	35.388	sandmark	128.993	springframework	624.388
squirrel_sql	9.082	struts	261.537	sunflow	27.408
tapestry	182.151	tomcat	352.572	trove	9.768
velocity	70.804	wct	99.622	webmail	18.074
weka	496.737	xalan	354.578	xerces	237.555
xmojo	43.249			**Total**	**33.971.977**

Table 5.3: The distribution of projects on the respective programming languages and their lines of code for the collection of open source projects.

Lang.	#	LOC	Lang.	#	LOC	Lang.	#	LOC
ABAP	10	55.498	Ada	5	194.961	C	145	4.791.612
COBOL	3	58.968	Delphi	3	15.470	Gosu	6	2.738
Groovy	66	143.939	Java	230	4.879.454	JavaScript	138	6.828.723
Kotlin	58	87.132	Matlab	30	134.229	OScript	122	2.837.202
PHP	68	1.016.013	Python	133	858.223	Rust	35	1.248.308
Swift	55	457.629	XML	13	28.541	**Total**	**1122**	**23.638.640**

5.4 Procedure

This section justifies the decisions we made for the different thresholds as well as our decision to highly multithread the different steps used in the approach.

5.4.1 Testing environment

The benchmarks are performed on an *Intel Core i7-6700HQ CPU* running with a frequency of *2.60GHz* on *four* physical cores with *40GB RAM*. To run the Java Code we used the *Eclipse IDE* executed on *Ubuntu 18.04.1 LTS (64-bit)* based on the *4.15.0-33-generic* Kernel.
We compare the durations of the single steps performed during the suffix-tree clone-detection approach to find generator-patterns. We used the projects *azureus, batik, checkstyle, cobertura, compiere, derby, drjava, exoportal, freecol, freecs, galleon, hsqldb, htmlunit, ireport, ivatagroupware, jFin_DateMath, javacc, jedit, jgrapht, jhotdraw, jmoney, joggplayer, jparse, jspwiki, jstock, jung, maven, netbeans, openjms, oscache, quilt, sandmark, squirrel_sql, tapestry, trove, weka, xerces* as a reference benchmark environment and ran the procedure several times to calculate a solid average value for the durations.
The decision for this environment is reasoned by its size of 37 projects including 60.235 source code files, whereas 3216 files are generated. It contains a total of 1.166.654 comments with a total of 23.683.887 valuable words. This pushes the machine the benchmark has been performed on to its maximum heap space it can provide to the Java Virtual Machine. The projects are chosen randomly to provide a meaningful mean of projects to preserve the generality of the Qualitas Corpus.

5.4.2 Thresholds

The threshold that has a direct impact on the number of found generator-patterns is the minimum clone length. It describes the minimal length a sequence of `CloneChunk`s has to provide to be considered by the Step 4.6. If a sequence is shorter than the minimum clone length it will not be added to the list of clone classes independent of the size the clone class would have.
Finding the best minimum clone length has been done by performing tests with different clone lengths ranging from a length of 2[1] up to a length of 25.
We finally decided on the minimum clone length of 5, where as the choice has been a mostly subjective one. Nonetheless did we consider the draw-off between the amount of results that are lost due to a to high minimum clone length and the amount of irrelevant data generated by a to low minimum clone length.
Figure 5.1 shows all found `CloneResult`s associated to the minimum clone length used to find it. It can be seen that by increasing the minimum clone length the number of `CloneResult`s drops nearly exponentially.
After applying the processing steps described in Step 4.7 to reduce the amount of data that is created, the remaining `CloneResult`s behave as shown in Figure 5.2. It shows that by filtering the amount of `CloneResult`s drops by around 99%, dropping even further by accumulating and clustering.
We tried using the minimum clone length of 3 at first due to the peak in the remaining `CloneResult`s. Nonetheless did we decide to use 5 because the observation was that no generator-patterns were lost by using this minimum clone length, but the manual search for the patterns in the links was much easier because the sequences were more meaningful.

5.4.3 Benchmarking

As displayed in Figure 5.3, the time-intensive steps are multithreaded.
We performed the benchmark with a minimum clone length of 5.
This results in time savings ranging from 5.32% *(Get Uniform Paths)* up to 86.25% *(Create Links For: GENERATED)*. The overall time saving sums up to a total of 56.5%, which reduces the average absolute amount of time of around 300 seconds down to 130 seconds.

[1] We didn't start at 1 because single words aren't meaningful when searching for generator-patterns. Especially words like *the, a, by ...* would have many occurrences that generate a huge amount of false positive clone classes.

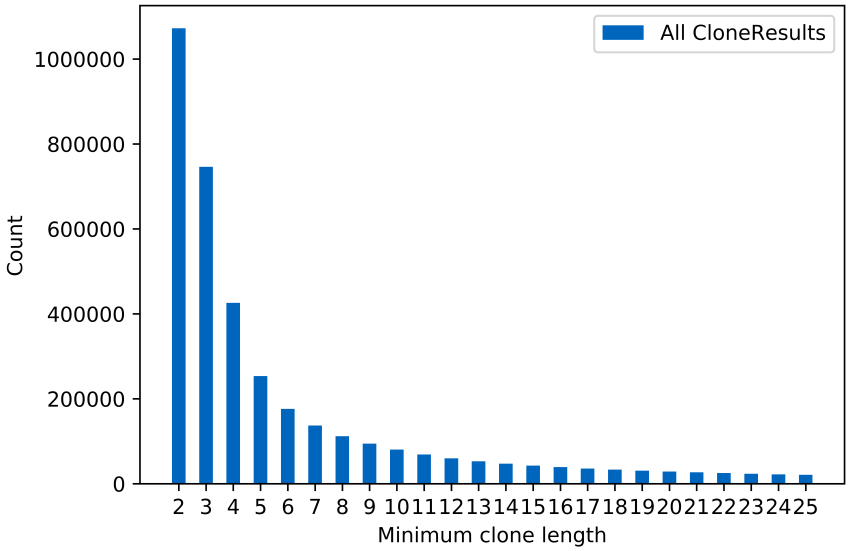


Figure 5.1: The number of `CloneResult`s found by the suffix-tree clone-detection in relation to the minimum clone length used to find them.

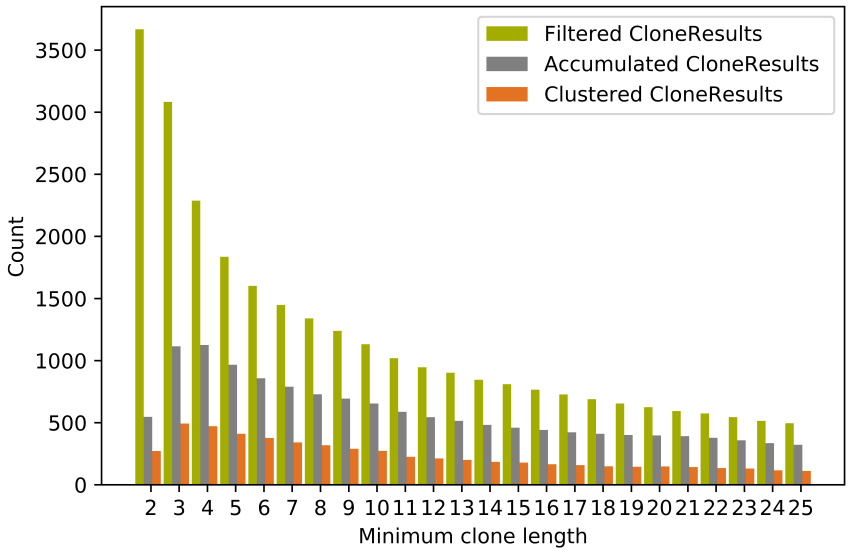


Figure 5.2: The number of `CloneResult`s that are left after the three processing steps described in Step 4.7 . The counts are associated to their respective minimum clone length used to find the `CloneResult`s.

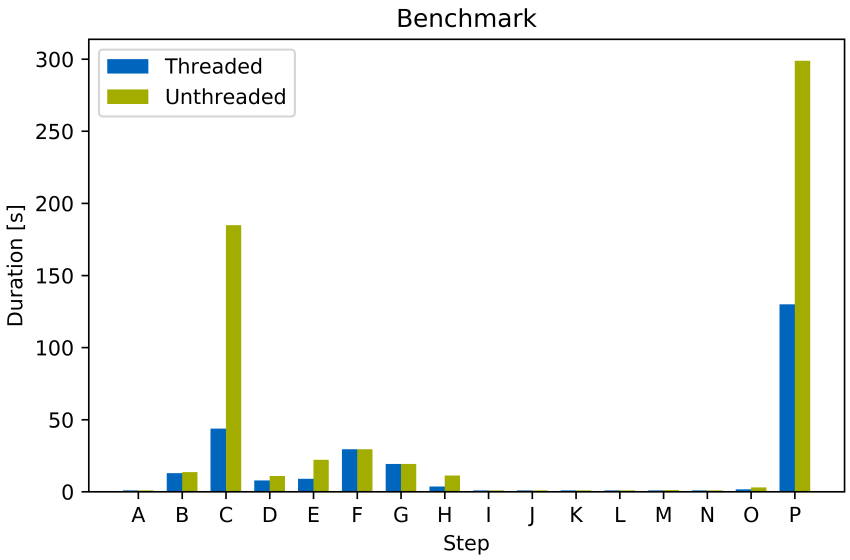


Figure 5.3: The results of the benchmarks performed during the search for generator-patterns. The step labels are explained in Table 5.5.

Table 5.5: Explanation for the step labels of Figure 5.3. The *Savings* column displays the time saving gained through multithreading.

Tick	Step	Savings	Tick	Step	Savings
A	Get Projects	-	B	Get Uniform Paths	5.32%
C	Retrieve Comments	76.29%	D	Save Comments	27.78%
E	Prepare Comments	59.29%	F	Build Suffix Tree	-
G	Find Clones	-	H	Convert to `CloneResult`s	67.54%
I	Filter Presumable Generator Patterns	-	J	Accumulate Presumable Generator Patterns	-
K	Cluster Presumable Generator Patterns	-	L	Filter Presumable Not Generator Patterns	-
M	Save Files For: GENERATED	83.06%	N	Create Links For: GENERATED	86.25%
O	Create Links For: NOT_GENERATED	42.41%	P	Total	56.5%

5.5 Results

This section displays the results we were able to create by using the approach to find generator-patterns and by using the the resulting generator-pattern repository on the different data sets.

5.5.1 Qualitas Corpus

At first we considered the Qualitas Corpus because it provided a good database and the results can be compared to the ones made in [6].
We found a total of 41 from the 112 projects to contain generated code.

Number of generator-patterns found

A total of 48 generator-patterns have been found. They are generated by 29 different generators. Additionally most of the patterns are regular expressions that accept up to 32 different patterns so that the final number of found generator patterns is much higher.

Lines of code for generated and manually maintained source code.

The Table 5.7 shows how many *lines of code* fall into the categories of generated and manually maintained code augmented with the ratios the categories cover. The ratio in lines of code for generated source code range from very low values *(0,05% - Hibernate)* up to more than three fourth *(75,03% - Cobertura)* of the overall lines of code.
The average proportion of generated lines of code is calculated using the equation

$$LOC_{avg} = \frac{\sum^{projects} \frac{LOC_{generated}}{LOC}}{|projects|} = 13.95\% \tag{5.1}$$

The total proportion of generated lines of code is calculated via

$$LOC_{total} = \frac{\sum^{projects} LOC_{generated}}{\sum^{projects} LOC} = 7.96\% \tag{5.2}$$

The average and total proportions of manually maintained lines of code are described by the complementary probability.

Number of generated and manually maintained source code files.

Table 5.9 additionally displays the *number of source code files* that are generated next to the ones that are manually maintained. The ratios the code categories cover are again associated to the absolute values, whereas they range from a very small amount of files *(0,02% - Hibernate)* up to around two third *(68,18% - SableCC)*.
Analog to the calculation of the total and the average ratios of generated to manually maintained lines of code as described in in Section 5.5.1 are the proportions of generated to manually maintained files calculated. The same equations 5.1 and 5.2 are used, but LOC is replaced by the number of files. This resulted in the proportions of generated files of

$$Files_{avg} = 8.07\%, \quad Files_{total} = 5.19\% \tag{5.3}$$

Again, the average and total proportions of manually maintained files are described by the complementary probability.

Lines of code per source code file

When looking at the Tables 5.9 and 5.7 we saw that in many projects only a few generated files are responsible for a high number in lines of code.
We decided to calculate the proportion of lines of code per file for the generated and the manually maintained files and could confirm that there exists indeed a correlation between the file size and the possible generation as stated in [6].
We calculated a **total** ratio of generated lines of code to generated files of

$$\frac{LOC_{generated}}{Files_{generated}} = 321, \quad \frac{LOC_{manually}}{Files_{manually}} = 204 \tag{5.4}$$

and an **average** ratio of

$$\frac{LOC_{generated}}{Files_{generated}} = 694, \quad \frac{LOC_{manually}}{Files_{manually}} = 214 \tag{5.5}$$

Table 5.7: The absolute and relative proportions of lines of code in the code categories generated and manually maintained in the Qualitas Corpus.

	Lines of Code				
Project	All	Generated		Manually	
argouml	389.952	11.992	3,08%	377.960	96,92%
axion	41.862	5.417	12,94%	36.445	87,06%
castor	349.301	18.056	5,17%	331.245	94,83%
cayenne	341.902	15.528	4,54%	326.374	95,46%
cobertura	68.154	51.139	75,03%	17.015	24,97%
compiere	727.702	206.999	28,45%	520.703	71,55%
derby	1.138.858	46.974	4,12%	1.091.884	95,88%
eclipse_SDK	4.956.920	103.292	2,08%	4.853.628	97,92%
exoportal	146.947	3.353	2,28%	143.594	97,72%
findbugs	185.806	5.895	3,17%	179.911	96,83%
galleon	135.442	4.888	3,61%	130.554	96,39%
gt2	1.514.789	325.259	21,47%	1.189.530	78,53%
hadoop	444.593	9.475	2,13%	435.118	97,87%
hibernate	711.370	327	0,05%	711.043	99,95%
hsqldb	269.978	1.100	0,41%	268.878	99,59%
ireport	338.819	100.276	29,60%	238.543	70,40%
jag	28.957	10.326	35,66%	18.631	64,34%
jasperreports	347.502	4.978	1,43%	342.524	98,57%
javacc	35.145	8.509	24,21%	26.636	75,79%
jboss	968.808	8.173	0,84%	960.635	99,16%
jchempaint	372.743	386	0,10%	372.357	99,90%
jedit	176.672	9.575	5,42%	167.097	94,58%
jena	164.679	5.045	3,06%	159.634	96,94%
jhotdraw	133.830	17.062	12,75%	116.768	87,25%
jparse	32.270	20.738	64,26%	11.532	35,74%
jrefactory	301.940	46.449	15,38%	255.491	84,62%
jruby	244.774	4.033	1,65%	240.741	98,35%
jstock	74.361	30.446	40,94%	43.915	59,06%
lucene	611.422	60.330	9,87%	551.092	90,13%
mvnforum	172.855	32.103	18,57%	140.752	81,43%
myfaces_core	189.954	19.678	10,36%	170.276	89,64%
netbeans	7.142.778	688.961	9,65%	6.453.817	90,35%
pmd	80.971	27.671	34,17%	53.300	65,83%
poi	363.487	21.621	5,95%	341.866	94,05%
sablecc	35.388	21.134	59,72%	14.254	40,28%
struts	261.537	6.978	2,67%	254.559	97,33%
tomcat	352.572	7.356	2,09%	345.216	97,91%
velocity	70.804	9.092	12,84%	61.712	87,16%
wct	99.622	150	0,15%	99.472	99,85%
weka	496.737	5.132	1,03%	491.605	98,97%
xalan	354.578	3.217	0,91%	351.361	99,09%

Table 5.9: The absolute and relative proportions of source code files in the code categories generated and manually maintained in the Qualitas Corpus.

	Source code files				
Project	All	Generated		Manually	
argouml	1.922	3	0,16%	1.919	99,84%
axion	237	7	2,95%	230	97,05%
castor	2.229	61	2,74%	2.168	97,26%
cayenne	2.796	227	8,12%	2.569	91,88%
cobertura	140	34	24,29%	106	75,71%
compiere	2.508	681	27,15%	1.827	72,85%
derby	2.842	17	0,60%	2.825	99,40%
eclipse_SDK	22.263	235	1,06%	22.028	98,94%
exoportal	1.839	23	1,25%	1.816	98,75%
findbugs	1.062	7	0,66%	1.055	99,34%
galleon	255	19	7,45%	236	92,55%
gt2	7.097	1.493	21,04%	5.604	78,96%
hadoop	1.988	14	0,70%	1.974	99,30%
hibernate	6.214	1	0,02%	6.213	99,98%
hsqldb	514	1	0,19%	513	99,81%
ireport	1.626	235	14,45%	1.391	85,55%
jag	132	20	15,15%	112	84,85%
jasperreports	1.589	10	0,63%	1.579	99,37%
javacc	195	40	20,51%	155	79,49%
jboss	6.806	199	2,92%	6.607	97,08%
jchempaint	2.042	5	0,24%	2.037	99,76%
jedit	531	11	2,07%	520	97,93%
jena	914	18	1,97%	896	98,03%
jhotdraw	613	67	10,93%	546	89,07%
jparse	75	7	9,33%	68	90,67%
jrefactory	2.229	140	6,28%	2.089	93,72%
jruby	1.128	9	0,80%	1.119	99,20%
jstock	274	52	18,98%	222	81,02%
lucene	3.036	89	2,93%	2.947	97,07%
mvnforum	705	100	14,18%	605	85,82%
myfaces_core	1.052	60	5,70%	992	94,30%
netbeans	32.567	1.864	5,72%	30.703	94,28%
pmd	751	162	21,57%	589	78,43%
poi	2.007	14	0,70%	1.993	99,30%
sablecc	198	135	68,18%	63	31,82%
struts	1.836	43	2,34%	1.793	97,66%
tomcat	1.324	44	3,32%	1.280	96,68%
velocity	371	9	2,43%	362	97,57%
wct	638	1	0,16%	637	99,84%
weka	1.271	9	0,71%	1.262	99,29%
xalan	975	2	0,21%	973	99,79%

5.5.2 Random projects

We started the search for generated code in the huge, randomly composed collection of open source projects by again applying the algorithm created for this thesis. In the first iteration we filtered the source code files using the generator-patterns found in the Qualitas Corpus. In the second iteration we used all generator-patterns found in the collection.
The Tables 5.11 and 5.13 show the results of the first and the second iteration. The column *Only Qualitas Corpus* refers to the first iteration using only the generator-patterns found in the Qualitas Corpus to filter the files. The column *Git Patterns added* describes the second iteration using all patterns found in the Qualitas Corpus and the open source projects.

Number of generator-patterns found

We found a total of 34 new generator patterns that are widely distributed over the different programming languages. These are again regular expressions that accept a multitude of different patterns so that the final number of found generator patterns is much higher.

Lines of code for generated and manually maintained source code.

Table 5.11 shows the results for the generated lines of code gathered during the two iteration steps.
In the first iteration the proportion of generated lines of code range from 0.01% for the Swift projects up to 8.88% for the Java projects. This result can be explained as the Qualitas Corpus contains only projects written in Java, so the previously found generator-patterns do apply best for the Java projects. Nonetheless some of the patterns are universal and are created by code generators that are able to generate code for different programming languages.
The total and average proportions for the lines of code are calculated analog to the procedure for the Qualitas Corpus and result in[2]:

$$LOC_{avg} = 2.60\%, \quad LOC_{total} = 3.23\% \tag{5.6}$$

Again, the average and total proportions of manually maintained files are described by the complementary probability.
In the second iteration the proportion of generated code found by using the patterns

[2] We include only the projects that do contain generated code so the results are better comparable to the ones gathered for the Qualitas Corpus.

included in the collection raises enormously. Especially in the projects not written in Java the amount of detected generated code is utterly enlarged.
Again we calculated the total and average proportions and found the following[2]:

$$LOC_{avg} = 17.66\%, \quad LOC_{total} = 12.88\% \tag{5.7}$$

Again, the average and total proportions of manually maintained files are described by the complementary probability.

Number of generated and manually maintained source code files.

Table 5.13 shows the results for the generated lines of code gathered during the two iteration steps.
The total and average proportions for the amount of generated source code files are again calculated using the equations used for the Qualitas Corpus. For the first iteration this results in the proportions of generated files of

$$Files_{avg} = 1.32\%, \quad Files_{total} = 2.15\% \tag{5.8}$$

For the second iterations the results are

$$Files_{avg} = 8.97\%, \quad Files_{total} = 8.68\% \tag{5.9}$$

Lines of code per source code file

As a last point concerning the open source projects we calculated again the ratio of lines of code per source code file for the two iterations.
In the first iteration we found the total ratios of of:

$$\frac{LOC_{generated}}{Files_{generated}} = 329, \quad \frac{LOC_{manually}}{Files_{manually}} = 217 \tag{5.10}$$

and an **average** ratio of

$$\frac{LOC_{generated}}{Files_{generated}} = 1948, \quad \frac{LOC_{manually}}{Files_{manually}} = 228 \tag{5.11}$$

In the second iteration the ratios changed to:

$$\frac{LOC_{generated}}{Files_{generated}} = 293, \quad \frac{LOC_{manually}}{Files_{manually}} = 215 \tag{5.12}$$

and an **average** ratio of

$$\frac{LOC_{generated}}{Files_{generated}} = 522, \quad \frac{LOC_{manually}}{Files_{manually}} = 198 \tag{5.13}$$

Table 5.11: The absolute and relative proportions of lines of code in the code categories generated and manually maintained in the collection of open source projects.

	Lines of Code				
Project	All	Only Qualitas Corpus		Git Patterns added	
ABAP	55.498	0	0,00%	0	0,00%
Ada	194.961	0	0,00%	42.312	21,70%
C	4.791.612	117.033	2,44%	402.644	8,40%
COBOL	58.968	0	0,00%	0	0,00%
Delphi	15.470	0	0,00%	0	0,00%
Gosu	2.738	0	0,00%	0	0,00%
Groovy	143.939	0	0,00%	386	0,27%
Java	4.879.454	433.357	8,88%	687.973	14,10%
JavaScript	6.828.723	27.151	0,40%	323.986	4,74%
Kotlin	87.132	0	0,00%	15.586	17,89%
Matlab	134.229	0	0,00%	0	0,00%
OScript	2.837.202	86.852	3,06%	449.116	15,83%
PHP	1.016.013	8.214	0,81%	191.794	18,88%
Python	858.223	0	0,00%	284.733	33,18%
Rust	1.248.308	0	0,00%	17.795	1,43%
Swift	457.629	52	0,01%	264.650	57,83%
XML	28.541	0	0,00%	0	0,00%

Table 5.13: The absolute and relative proportions of source code files in the code categories generated and manually maintained in the collection of open source projects.

	Source code files				
Project	All	Only Qualitas Corpus		Git Patterns added	
ABAP	228	0	0,00%	0	0,00%
Ada	1.433	0	0,00%	232	16,19%
C	11.108	19	0,17%	265	2,39%
COBOL	161	0	0,00%	0	0,00%
Delphi	15	0	0,00%	0	0,00%
Gosu	15	0	0,00%	0	0,00%
Groovy	1.499	0	0,00%	4	0,27%
Java	32.573	1.522	4,67%	2.971	9,12%
JavaScript	25.461	6	0,02%	347	1,36%
Kotlin	929	0	0,00%	106	11,41%
Matlab	1.408	0	0,00%	0	0,00%
OScript	18.076	481	2,66%	2.462	13,62%
PHP	4.847	16	0,33%	1.344	27,73%
Python	3.035	0	0,00%	204	6,72%
Rust	3.636	0	0,00%	26	0,72%
Swift	2.638	2	0,08%	1.173	44,47%
XML	195	0	0,00%	0	0,00%

5.6 Discussion

By analyzing the projects included in the Qualitas Corpus we found a wide variety of generator-patterns and were able to calculate the average and total proportions of generated to manually maintained lines of code and files. We saw that generated code makes only a small portion of the all lines of code, whereas in average a project contains 13.95% generated lines of code and all projects contained 7.96% generated lines of code. Additionally we saw that in average 8.07% of files are generated whereas in total only 5.19% are generated.
We were able to find even more generator-patterns and resulting a higher number of generated source code after applying the algorithm on the huge set of open source projects. Hereby we found that the projects do contain even more generated code with a total of 12.88% and an average of 17.66% of the lines of code being generated. Also the total amount of files has risen to 8.97% and the total amount to 8.68%.
Resulting from that we showed that in average the generated files always contain more lines of code than the manually maintained ones, whereas there exists no direct correlation of the actual lines of code ratios between generated and manually maintained source code files.

5.6.1 Completeness and accuracy

We found the generator-patterns by iterating over the Qualitas Corpus and applying the algorithm described in Chapter 4. The found patterns have been added to the generator-pattern repository and used to filter all files that we were able to classify as generated in the next iteration.
In every iteration the amount of possible generator-patterns could be lowered by using this technique very fast until all generator-patterns were reviewed by hand and so all patterns have been found in the Qualitas Corpus.
We compared our results to these described in [6] and could confirm that our approach found all generated files. Resulting from this we concluded that our approach is as complete as it can be using the technique of suffix-tree clone-detection on the comments.
By using the approach on the huge random collection of open source projects we detected some more generator-patterns that were not included in the Qualitas Corpus. Nonetheless we were able to filter many generated files in the first iteration by using the generator-pattern repository what supports our assertion that the algorithm found all generator patterns that were included in the Qualitas Corpus.
Additionally, the generator-pattern repository is easily extendable with new patterns at any time what was shown by applying the algorithm on the collection of open

source projects. So by gradually extending the repository by applying the suffix-tree clone-detection approach on the comments, the repository will saturate with patterns and will over time become complete and highly accurate in terms of generator-patterns.

5.6.2 Relevance of results

Not sure what to write in this section.
Maybe how the generator-pattern repository can be used by Teamscale to lower amount of work done in analysis by excluding the generated files?

5.7 Threats to validity

This section describes the threads that may have an impact on the validity of the results generated by our approach and justifies how .

5.7.1 Wrong filtering

In Step 4.7.1 we used the regular expression shown in Listing 4.4. This reduces the amount of resulting `CloneResult`s to a level that can be processed by a human. Our observations showed that indeed most of the generator-patterns contained the regex so that it can be considered as only a small thread.
Nonetheless did we find generator-patterns that didn't contain the regex. These patterns make only a very small portion of all generator-patterns so we didn't see a value in refining it. Furthermore does step 4.8 also save files and links for the not generated found `CloneClass`es that can be reviewed in addition to the generated ones. The problem arising is that the amount of data to review is way to high as i could be process by hand, but the ones that had the highest number of occurrences were also searched by us what revealed the patterns that didn't include the regular expression.
In conclusion we can say that the regular expression is a very good filter to find the generator-patterns which can be easily extended to also review the patterns that were filtered out to find all generator-patterns.

5.7.2 Minimum clone length vs. irrelevant data

We faced the decision on a most optimal minimum clone length while applying the suffix-tree clone-detection algorithm made by Esko Ukkonen [4] and the search for clones on the resulting tree structure. We finally decided on a length of 5.
The draw-off we faced was the one that a high minimum clone length resulted in a low number of `CloneResult`s. Hereby were all short generator-patterns sorted out before

we were even aware that they existed in the code base.
On the other hand was a low minimum clone length not really much applicable due to the fact that it generated really much false positives like only the words *generated, modify, edit*, whereas they do not hold any value without the context they were used in. Especially the word *generated* is often used in the context that the following method generates some sort of date, not that the method itself is generated.
Therefore we did a benchmark iterating over the minimum clone length and comparing the amount of data the different lengths generated and the subjective observation how useful the data was. In Figures 5.1 we saw that the unfiltered `CloneResult`s do follow a nearly exponential descend, whereas a the minimum clone length of 5 creates only around 15% compared to a length of 2. This arises the question whether a minimum clone length of 5 erases to much valuable data.
When you look at Figure 5.2 we saw that by applying the three processing steps of filtering by the regular expression described in Step 4.7.1, then accumulating the `CloneResult`s as stated in Step 4.7.2 and finally clustering the results explained in Step 4.7.3 resulted in an amount of found `CloneResult`s having it's peak at the minimum clone length of 3. Nonetheless we decided on a length of 5 due to the fact that it erases only around 20% in the final presumable generator-patterns generated, but it generates patterns that are much more speaking for themselves due to the extended length.
Additionally, we observed that the patterns which are lost by increasing the minimum clone length up to the length of 5 are all included in the patterns we found also by using the higher length. This is due to the fact that the shorter presumable patterns do overlap with the patterns found with the length of 5, but are no direct substrings of the longer generator-patterns. Nonetheless they do point to the same classes and in fact to the same patterns.
Following these thoughts we decided on a minimum clone length of 5.

5.7.3 Representativeness of data sets

We used the Qualitas Corpus as a reference set because it "reduces the cost of performing large empirical studies of code and supports comparison of measurements of the same artifacts."[1, p. 1] It is used in a variety of other studies and the results can be easily compared.
As we showed it includes a high number of generated code and also many generator-patterns, what makes it a good starting point to begin working with the created algorithm. Furthermore it consists of a high number of different projects (112) that are all maintained by big software companies and used in production. Due to the fact that *CQSE GmbH* and their product *Teamscale* is mostly used by big international companies

to monitor their code quality, the parallels found between the projects led us to the conclusion that the Qualitas Corpus is a good starting point for the use in *Teamscale*.
Nonetheless the problem we see in using the Qualitas Corpus is that it consists only of projects written in Java, which makes it only representative for this particular programming language. To avoid this problem we additionally used the algorithm on a huge randomly composed collection of open source projects we gathered from GitHub. This dataset is composed of very small up to really big projects that are written in a multitude of programming languages and developed by programmers all around the globe.
This led us to the conclusion that in conjunction the two reference datasets give a representative profile of very much, very different projects that are all developed under very unique circumstances.

5.7.4 Generators without pattern

A last thread we are facing is that there exists also a multitude of code generators that do not mark their artifacts with a characteristic pattern in the comments. Especially custom written generators often don't add a generator pattern.
This problem can't be faced with the current approach of suffix-tree clone-detection on the comments. To find these generated artifacts one has to combine our approach with e.g. the one described in [6].

6 Future Work

We discussed different possible future tasks that build up on the work done so far. These tasks are extensions and possible enhancements of the presented approach and give possibilities to bring the work to a state where it is usable in production.

6.1 Integration into Teamscale

The preliminary task is to integrate the generator-pattern repository into Teamscale. Therefore we have to enhance the current implementation for exclude patterns so that the generator-pattern repository can automatically detect the generated source code files during the initial analysis of the uploaded projects. This could be done fully automatic or in a way that the user has to choose for each project or even each file if it should be excluded if detected as generated.
As we have seen in the previews chapters, the amount of generated code in average ranges from around 5% up to 20%. By including the generator-pattern repository into the static code analysis done by Teamscale the amount of work could be lowered drastically. Also execution time and the number of findings would be much lower what would make it easier to focus on quality management tasks that really have value rather than being bothered by findings that concern generated code.
Our observation has been that code generation is often used in the field of lexical analysis and parsing. The resulting classes contain often many, very deep nested conditional statements. These generate many findings concerning metrics like the cyclomatic complexity. Furthermore is generated code always generated using templates or predefined grammars which leads finished classes to be mostly clones of each other, no matter when and in what context the source code files were generated. This leads to a high number of clones among the projects and to a high number of findings.
So to speed up and to lower the amount of unnecessarily generated findings a future task is to integrate the generator-pattern repository into Teamscale.

6.2 Granularity of the generator-pattern repository

Currently, the generator-pattern repository can only search for the generator-patterns in the source code files. If one pattern is found in a file it is considered as generated. This may be applicable in most of the cases, but we also observed patterns that only mark the following method or datatype definition as generated. This may lead to false positives in the detection of generated code.
To avoid this a future task is to refine the granularity of the generator-pattern repository by adding additional information about the scope the pattern concerns.
This would enhance the accuracy of the repository and would make it even better in the detection of generated source code.

6.3 Different clone detection approaches

We currently use the suffix-tree clone-detection approach made by Esko Ukkonnen [4]. This approach is linear in time and space and thus good scalable.
Nonetheless is the algorithm based on a recursive approach what makes it not parallelizable. Furthermore is the approach only capable of finding exact clones in strings. By extending it to an approximate string matching approach the performance could be enhanced.

6.3.1 Approximate string matching approaches

We observed that many of the generator-patterns do differ in single words, signs or the time used in the pattern. These small differences arise through changes in the templates used in the different versions of the generator.

- It is probably a good idea to avoid changing this class manually, since it may be overwritten next time code is regenerated.
- It is probably a good idea to avoid changing this class manually, since it will be overwritten next time code is regenerated.

The problem we faced is that e.g. these two generator-patterns do not occur as one clone class, what surely is intended by the algorithm of [4]. Nonetheless could it be a nice future task to implement the approximate string matching algorithm described in [5] to have these clone classes fall into one class what could improve the search for generator-patterns and maybe even make it completely automatic.

6.3.2 Parallelizable approaches

Many steps of the approach used in this thesis are highly multithreaded to speed up the processing. For benchmarks see 5.4.3.
As we have seen the building of the suffix-tree and the finding of clones take around 45% of the total time the algorithm runs.
This made us think about parallelizing these two steps to increase the speed of the algorithm especially when processing large scale code bases.
In the work presented in [8] a parallel and efficient approach to large scale clone detection is presented that uses a new token-based approach. A filtering heuristic is presented that reduces the amount of token comparisons. This heuristic is used in a MapReduce based parallel algorithm that easily scales to thousands of projects.
We came to the conclusion that by implementing this algorithm or one using another parallelizable approach could increase the efficiency of our algorithm.

6.4 Evaluation on more study objects to find more generator patterns

The goal of the generator-pattern repository is to provide a database that can be used to detect generated source code efficient and reliable. As we have already seen in the transition from the Qualitas Corpus to the huge, randomly composed set of open source projects we could improve the capabilities of the generator-pattern repository by far.
By applying the algorithm on even more projects in the future we can enhance the efficiency increasingly, and maybe to the point where it detects all code generators that are in existence up to this day.

7 Conclusion

We presented an algorithm that uses a suffix-tree clone-detection approach to find clones among the comments in source code. By using this approach we implemented a semi-automatic way to build up a generator-pattern repository.
We used a Teamscale server that extracts the comments out of the source code files that have been uploaded. These comments are normalized to be usable in a modified suffix-tree algorithm. Based on the suffix-tree we searched for clones and filtered the results to make them processable manually. The filtered results have been saved and reviewed by hand to build up the generator-pattern repository. This repository holds a multitude of generator-patterns associated to their respective generator.
We used the algorithm on a reference data set (Qualitas Corpus) and a huge, randomly composed collection of open source projects we gathered from GitHub to find a wide range of generator-patterns for a multitude of generators and many different programming languages. Resulting we filled the repository with a total of 82 generator-patterns.
We used the found patterns on the data sets to categorize the source code in *generated* and *manually maintained* code. Hereby we found very different proportions of generated code ranging from 0% for some whole programming languages up to 75% for single projects. This resulted in a average of 5% to 20% for generated code in projects that use code generators.
In conclusion we saw that the suffix-tree clone-detection approach to find generator-patterns is promising in the search for generated code and that the generator-pattern repository is an efficient and reliable attempt for automatic code classification.

Listings

List of Figures

List of Tables

Bibliography

[1] J. Tempero, Ewan and Anslow, Craig and Dietrich, Jens and Han, Ted and Li, Jing and Lumpe, Markus and Melton, Hayden and Noble, "Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies," *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pp. 336–345, 2010. DOI: http://dx.doi.org/10.1109/APSEC.2010.46.

[2] T. L. Alves, "Categories of Source Code in Industrial Systems," *2011 International Symposium on Empirical Software Engineering and Measurement*, pp. 335–338, 2011, ISSN: 1938-6451. DOI: 10.1109/ESEM.2011.42.

[3] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," *Queen's School of Computing TR*, vol. 115, p. 115, 2007. DOI: 10.1.1.62.7869.

[4] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995, ISSN: 01784617. DOI: 10.1007/BF01206331.

[5] E. Ukkonen, "Approximate string-matching over suffix trees," *Combinatorial Pattern Matching*, pp. 228–242, 1993, ISSN: 0926-9630. DOI: 10.1007/BFb0029808.

[6] J. Bernwieser, "Automatic Categorization of Source Code in Open-Source Software," 2014.

[7] Apache Software Foundation, *Mahout*.

[8] H. Sajnani and C. Lopes, "A parallel and efficient approach to large scale clone detection," *2013 7th International Workshop on Software Clones (IWSC)*, vol. 7300, pp. 46–52, 2013. DOI: 10.1109/IWSC.2013.6613042.