DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor Thesis

Recognition of Generated Code in Open Source Software

Marcel Bruckner

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor Thesis

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:	16.08.2018

I confirm that this bachelor thesis is my own work and I have documented all sources and material used.

Munich, 16.08.2018 Marcel Bruckner

Acknowledgments

Abstract

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. **Details zum Datensatz**

- Automatic classification of source code in a variety of open source systems and evaluation of the amount of generated code. **paar Statistiken als Teaser einbringen (z.B. wieviele Projekte, wieviel code, sprachen)**

2 Terms and Definitions

2.1 Terminology

2.1.1 Generated code

Im Moment noch einfach kopiert. Dringend umschreiben.
As proposed in [1] 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.

2.1.2 Manually-maintained code

Im Moment noch einfach kopiert. Dringend umschreiben.
In contrast we consider *manually-maintained code* as suggested in [1] 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*.

2.1.3 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.

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 and the scope that the pattern denotes as generated.

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 [2].

2.1.6 Clone chunk

For the use in Ukkonens algorithm for the construction of a suffix-tree [3] 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.

2.1.9 Clone result

2.2 Metrics

2.2.1 Lines of code

Im Moment noch einfach kopiert. Dringend umschreiben.

Code snippets draus machen

Table 2.1: Clone Pair and Clone Class

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

As stated in [4] the *Lines of Code (LOC)* metric represents the size of a software and is thus one of the easiest ways to represent its complexity. Generally the LOC metric does not provide very meaningful results as code quality does not correlate with the number of lines; it's still useful in order to give an impression about a class' size and thus its respective impact on the overall quality.

2.2.2 Clone coverage

Im Moment noch einfach kopiert. Dringend umschreiben.
Clone coverage is an important metric to reflect the maintainability and the extendibility of a software. It defines the probability that an arbitrarily chosen statement is part of a clone. If the clone rate is too high, it can be very dangerous to implement changes as they have to be performed on every clone.

2.2.3 Comment ratio

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.

2.3.1 suffix-tree clone-detection

In [3], 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.

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.

2.3.5 Token type

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

3 Related Work

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 [4]. 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 dataset 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 LOC 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 true 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 My extension

Bessere Überschrift

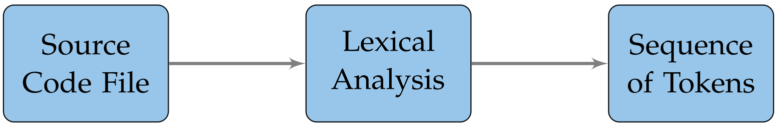
In this thesis, the approach differs from Bernwiesers so that the clone-detection is done on the comments added to the source code files in the dataset, rather than on the respective source code. A goal of our approach is to find the specific generator patterns that Bernwieser pointed at. It's based 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:

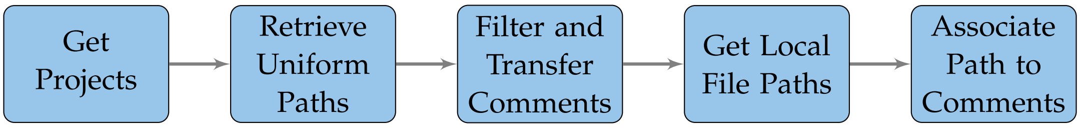
1. Get the projects that are currently available in the Teamscale instance.

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 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 [3][5].
The preparation is highly multithreaded to speed it up. For benchmarks see 5.4.3.

4.3.1 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.

4.3.2 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 2.3.3.

4.3.3 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.

4.3.4 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.

4.4 Build suffix-tree

Esko Ukkonen introduced an algorithm to construct suffix-trees on strings in his paper for *Algorithmica* in 1995 [3]. "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." [3]

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.

For a deeper insight into the details of the algorithm for suffix-tree construction see [3].

4.5 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

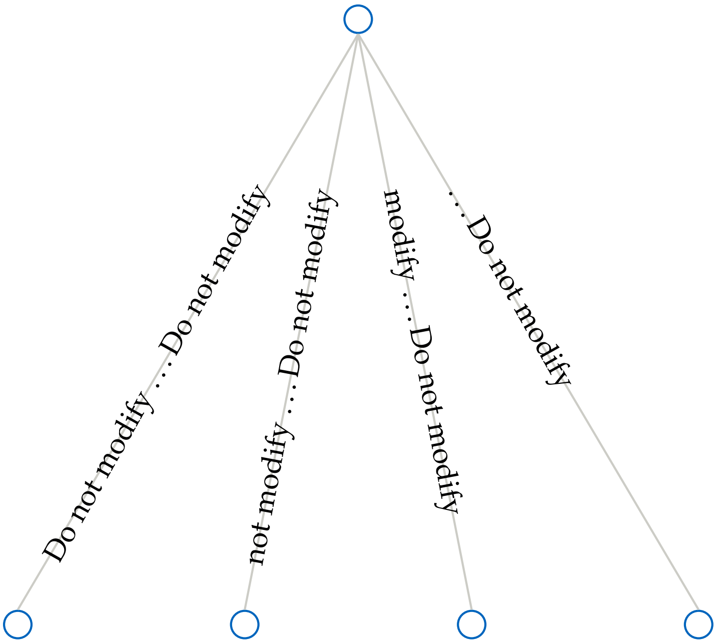

Do not modify . . . Do not modify
not modify . . . Do not modify
modify . . . Do not modify
. . . Do not modify

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

perform a breadth first search to find clones.
When the graph search is performed the clone classes get returned as a List whereas each entry represents a CloneClass. Each CloneClass itself is a List of List of CloneChunks. Each inner List of the CloneClass represents a clone instance. The clones consist of a List of CloneChunks 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 CloneChunks gets now converted into a List of CloneResult. A CloneResult is a more compact and convenient representation of the clone classes. 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)
```

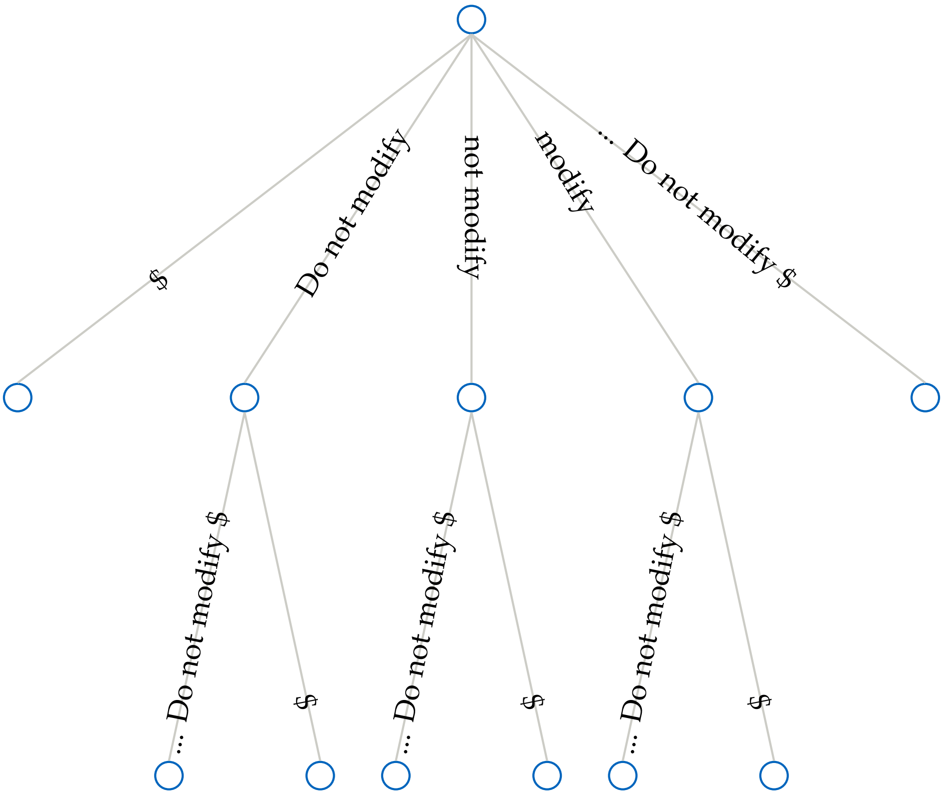


Figure 4.2: The explicit suffix tree for the list of words "Do not modify . . . Do not modify".

```
        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 CloneClasses 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"]],
}
```

4.6 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.

4.6.1 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 [4] 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)?)"
```

4.6.2 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 suspect that the sets of occurrences of the CloneResults do overlap in this case, 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 CloneResults 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 CloneResults before accumulation with a minimum clone length of 2.

```
{
        "name": "generated by", "occurrences": [
                {originID: "/projectX/class1.java", lineNumber: "9"}
                {originID: "/projectX/class2.java", lineNumber: "9"}
                {originID: "/projectX/class3.java", lineNumber: "9"}],
        "name": "is generated by", "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: "/projectX/class1.java", lineNumber: "9"}
                {originID: "/projectX/class2.java", lineNumber: "9"}
                {originID: "/projectX/class3.java", lineNumber: "9"}],
}
```

Listing 4.6: The CloneResults after accumulation with a minimum clone length of 2.

```
{
        "name": "generated by", "occurrences": [
                {originID: "/projectX/class1.java", lineNumber: "9"}
                {originID: "/projectX/class2.java", lineNumber: "9"}
                {originID: "/projectX/class3.java", lineNumber: "9"}]
}
```

4.6.3 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 CloneResults 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.

4.7 Generate links to the files

The last step performed by the tool is to save the CloneResults 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 CloneResults 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.

4.8 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 [4], 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 <a href="http://java.sun.com/xml/jaxb"> http://java.sun.com/xml/jaxb</a> 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 and reviewed randomly for their amount of false positives.
Bernwieser [4] 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 [4]. 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 and their importance in programming.

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 randomly composed collection of open source projects that is used to test the generator-pattern repository in a not particularly curated environment.

5.2.1 Qualitas Corpus

"The Qualitas Corpus is a curated collection of software systems intended to be used for empirical studies of code artefacts. The primary goal is to provide a resource that supports reproducible studies of software. The current release of the Corpus contains open-source Java software systems, often multiple versions." [6]
Nonetheless the procedure is designed to be also applicable onto sets including other programming languages as well as sets of projects with mixed ones.
The current release is from the year 2013 and includes 112 different projects. To be easily comparable to the work in [4], 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 Git 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 level of randomness and diversity for the projects. **Describe more**

5.3 Study Design

This section describes how the study, using the information from the study objects, attempts to answer the research questions.

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 [4] to derive the improvement our

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		

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 generated to manually maintained source code and files.
- Amount of generated code detected by our approach compared to the one used in [4].

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* **Cite** 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 3034 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 4.5. 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 4.6 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.

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.

[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.

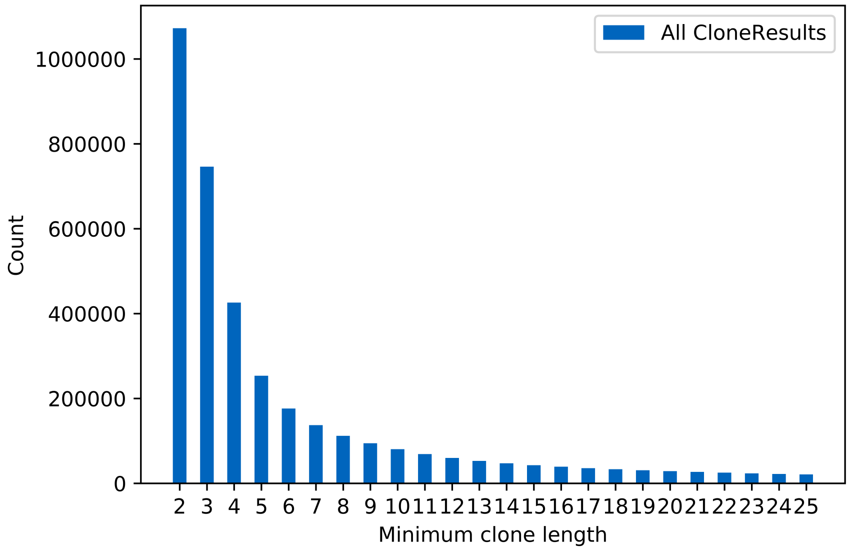


Figure 5.1: This chart shows the number of `CloneResults` found by the suffix-tree clone-detection in relation to the minimum clone length used to find them.

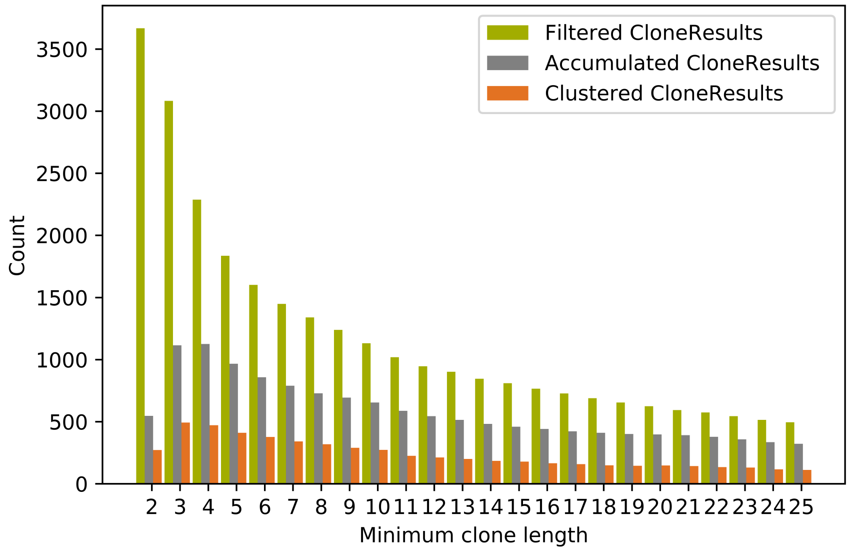


Figure 5.2: This chart shows the number of `CloneResults` that are left after the three processing steps described in Step 4.6. The counts are associated to their respective minimum clone length used to find the `CloneResults`.

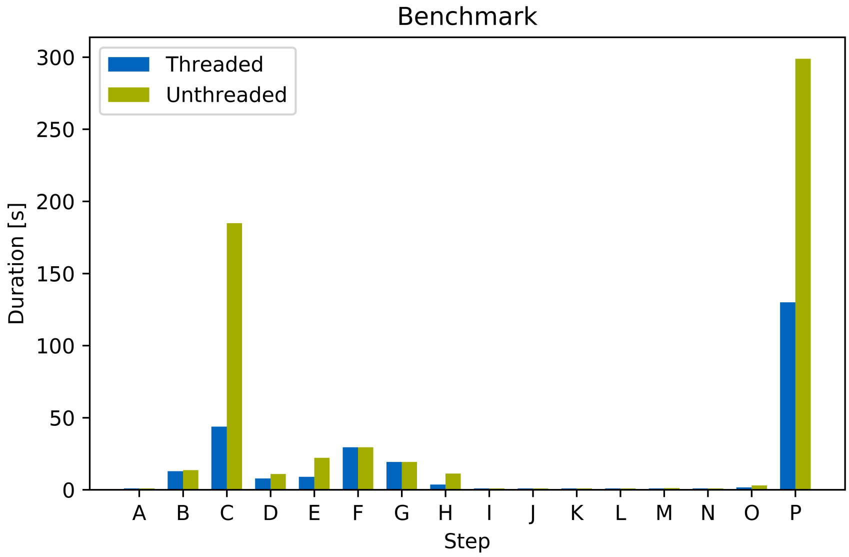


Figure 5.3: The results of the benchmarks performed during the search for generator-patterns. The step labels are explained in Table 5.3.

Table 5.3: 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%

5.5.1 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 [4].

Lines of code for generated and manually maintained source code.

-

Number of generated and manually maintained source code files.

-

Ratio of generated to manually maintained source code and files.

-

Amount of generated code detected by our approach compared to the one used in [4].

-

5.5.2 Random Git Projects

a

5.6 Discussion

a

5.6.1 Completeness and accuracy

a

5.6.2 Relevance of results

a

5.7 Threats to validity

a

Table 5.5: The results our algorithm has created.

Project	Lines of code				Comparison
	All	Generated	Manually	Ratio	
axion	41862	5417	36445	14,86%	
castor	349301	18056	331245	5,45%	
cayenne	341902	15473	326429	4,74%	
cobertura	68154	51139	17015	300,55%	
compiere	727702	204462	523240	39,08%	
derby	1138858	46974	1091884	4,30%	
eclipse_SDK	4956920	102546	4854374	2,11%	
exoportal	146947	3353	143594	2,34%	
findbugs	185806	5895	179911	3,28%	
galleon	135442	4888	130554	3,74%	
gt2	1514789	322319	1192470	27,03%	
hadoop	444593	9475	435118	2,18%	
hibernate	711370	327	711043	0,05%	
hsqldb	269978	1100	268878	0,41%	
ireport	338819	100276	238543	42,04%	
jag	28957	9457	19500	48,50%	
jasperreports	347502	3660	343842	1,06%	
javacc	35145	8509	26636	31,95%	
jboss	968808	7556	961252	0,79%	
jchempaint	372743	134	372609	0,04%	
jedit	176672	9575	167097	5,73%	
jena	164679	5045	159634	3,16%	
jhotdraw	133830	17062	116768	14,61%	
jrefactory	301940	46449	255491	18,18%	
jruby	244774	4033	240741	1,68%	
jstock	74361	30446	43915	69,33%	
lucene	611422	60330	551092	10,95%	
mvnforum	172855	32103	140752	22,81%	
myfaces_core	189954	19678	170276	11,56%	
netbeans	7142778	678217	6464561	10,49%	
pmd	80971	27671	53300	51,92%	
poi	363487	21621	341866	6,32%	
sablecc	35388	21134	14254	148,27%	
struts	261537	6978	254559	2,74%	
tomcat	352572	7356	345216	2,13%	
velocity	70804	9092	61712	14,73%	
wct	99622	150	99472	0,15%	
weka	496737	5132	491605	1,04%	
xalan	354578	3217	351361	0,92%	

Table 5.7: The results our algorithm has created.

Project	Source code files				Comparison
	All	Generated	Manually	Ratio	
axion	237	7	230	3,04%	
castor	2229	61	2168	2,81%	
cayenne	2796	226	2570	8,79%	
cobertura	140	34	106	32,08%	
compiere	2508	675	1833	36,82%	
derby	2842	17	2825	0,60%	
eclipse_SDK	22263	234	22029	1,06%	
exoportal	1839	23	1816	1,27%	
findbugs	1062	7	1055	0,66%	
galleon	255	19	236	8,05%	
gt2	7097	1487	5610	26,51%	
hadoop	1988	14	1974	0,71%	
hibernate	6214	1	6213	0,02%	
hsqldb	514	1	513	0,19%	
ireport	1626	235	1391	16,89%	
jag	132	18	114	15,79%	
jasperreports	1589	7	1582	0,44%	
javacc	195	40	155	25,81%	
jboss	6806	190	6616	2,87%	
jchempaint	2042	3	2039	0,15%	
jedit	531	11	520	2,12%	
jena	914	18	896	2,01%	
jhotdraw	613	67	546	12,27%	
jrefactory	2229	140	2089	6,70%	
jruby	1128	9	1119	0,80%	
jstock	274	52	222	23,42%	
lucene	3036	89	2947	3,02%	
mvnforum	705	100	605	16,53%	
myfaces_core	1052	60	992	6,05%	
netbeans	32567	1851	30716	6,03%	
pmd	751	162	589	27,50%	
poi	2007	14	1993	0,70%	
sablecc	198	135	63	214,29%	
struts	1836	43	1793	2,40%	
tomcat	1324	44	1280	3,44%	
velocity	371	9	362	2,49%	
wct	638	1	637	0,16%	
weka	1271	9	1262	0,71%	
xalan	975	2	973	0,21%	

5.7.1 Wrong filtering

a

5.7.2 Minimum clone length vs. irrelevant data

a

5.7.3 Representativeness of data sets

a

5.7.4 Generators without pattern

a

6 Future Work

- In Teamscale einbauen

7 Conclusion

List of Figures

List of Tables

Bibliography

[1] 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.

[2] 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.

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

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

[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. 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.

[7] Apache Software Foundation, *Mahout*.