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

Acknowledgments iii

Abstract iv

1 Introduction 1

2 Terms and Definitions 2
2.1 Terminology . . . 2
2.1.1 Generated code . . . 2
2.1.2 Manually-maintained code . . . 2
2.1.3 Generator pattern . . . 2
2.1.4 Generator pattern repository . . . 2
2.1.5 Clone . . . 3
2.1.6 Clone chunk . . . 3
2.1.7 Sentinel . . . 3
2.1.8 Clone class . . . 3
2.1.9 Clone result . . . 3
2.2 Metrics . . . 3
2.2.1 Lines of code . . . 3
2.2.2 Clone coverage . . . 4
2.2.3 Comment ratio . . . 4
2.3 Teamscale . . . 4
2.3.1 Suffix Tree Clone Detection . . . 5
2.3.2 Lexer . . . 5
2.3.3 Token . . . 5
2.3.4 Token class . . . 5
2.3.5 Token type . . . 5

3 Related Work 7
3.1 Automatic Categorization of Source Code in Open-Source Software - Jonathan Bernwieser . . . 7
3.1.1 Number of clones per clone class . . . 7
3.1.2 Clone coverage per class . . . 8

3.1.3 Results 8
3.1.4 My extension 8

4 Approach **9**
4.1 Use Teamscale as Lexer to extract Tokens 9
4.2 Connect to the Teamscale server and retrieve comments 9
4.3 Prepare comments for suffix tree clone detection 10
4.3.1 Create thread for each source code file 11
4.3.2 Remove unnecessary characters and whitespaces 11
4.3.3 Convert words into clone chunks 11
4.3.4 Add sentinels 12
4.3.5 Merge thread results 12
4.4 Build suffix tree 12
4.5 Find clones 13
4.6 Filter possibly generated clone results 16
4.7 Generate links to the files 17
4.8 Generation of a Generator-Pattern Repository 18

5 Evaluation **19**

6 Future Work **20**

7 Conclusion **21**

List of Figures **22**

List of Tables **23**

Bibliography **24**

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.
As stated in [4] the *Lines of Code (LOC)* metric represents the size of a software and is

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 . . . */

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 2.2.

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*.

Table 2.2: 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.

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

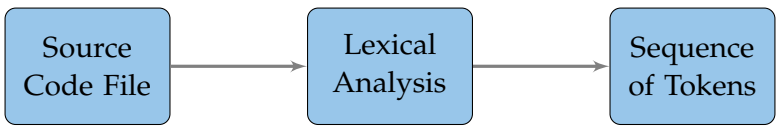
In this thesis, the approach differs from Bernwiesers so that the clone detection is done on the comments added to the source code files in the dataset, rather than on the respective source code. A goal of our approach is to find the specific generator patterns that Bernwieser pointed at. It's based 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 by means of the comments. Following a generator pattern repository has been created. The target of this repository is to provide a database of code generators and their respective characteristic generator pattern which identifies the source code containing it as generated.
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 used as a benchmark. 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 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.

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	...

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].

Table 4.2: Programming language specific regular expressions that represent comment identifiers (Escape characters are omitted).

Programming language(s)	Regular expression
C-like	((^(/** | /* | * | / /)) | (*/)$)
Ada	(^(–))
Matlab	(^(%))
Python	(^(#))
XML	((^(<!--)) | (-->)$)

4.3.1 Create thread for each source code file

The step 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 Remove unnecessary characters and whitespaces

At first the comments are split at the linebreaks into single lines and leading and trailing whitespaces of each line are removed.
Susequent the comment enclosings are removed using a regular expression on every line to remove the language specific identifiers of comments. This regular expression is build from a list of regular expressions that represent the specific comment enclosings in the respective language.
These are concatenated with the *OR*-operator to generate a generic regular expression for most programming language supported by Teamscale (2.3). Following this step the the leading and trailing whitespaces are removed again.
An overview of the regular expressions that are used in this step are shown in table 4.2.

4.3.3 Convert words into clone chunks

The resulting lines of the previous step are again split into single words. Every word is at first checked if it holds valuable information for the approach. To check the value of each word it gets checked if it contains alphabetic letters *(a-z, A-Z)* or digits. If a word only consists of nonalphabetic and no digits it gets sorted out *(e.g. "--------")*.

The remaining words that bring a value to the approach are now converted into clone chunks. Therefore is added to every word the type of the comment, the offset and the linenumber 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.

4.3.4 Add sentinels

To prevent the suffix tree clone detection algorithm from intermixing the comments a sentinel gets added at the end of the list of clone chunks resulting from the previous steps. This is necessary due to the fact that the algorithm works an a single list of clone chunks that is build from merging all clone chunks from every comment.

4.3.5 Merge thread results

Once all threads are finished, resulting in all comments being converted into lists of clone chunks and sentinels being added to the end of each list, all lists get concatenated into one hugh list containing all words of the comments. On this list the suffix tree clone detection algorithm can now be applied.

4.4 Build suffix tree

Im Moment noch einfach kopiert. Dringend umschreiben.
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. To construct it a string gets processed symbol by symbol from the start to the end. The construction is based on the observation is that the suffixes of $T^i = t_1 \dots t_i$ can be obtained from catenating t_i to the end of each suffix of $T^{i-1} = t_1 \dots t_{i-1}$.
For this thesis the algorithm has been expanded to work on any arbitrary datastructure. The simple observation behind this 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 element is added by following one of three rules:

1. Walk all paths and add the new element.
2. If a path does not exists, add it.
3. If a path already exists, do nothing.

This property of the algorithm has been used to extend it to a generic version of it by allowing any list of data. The only mandatory property the data structure must provide

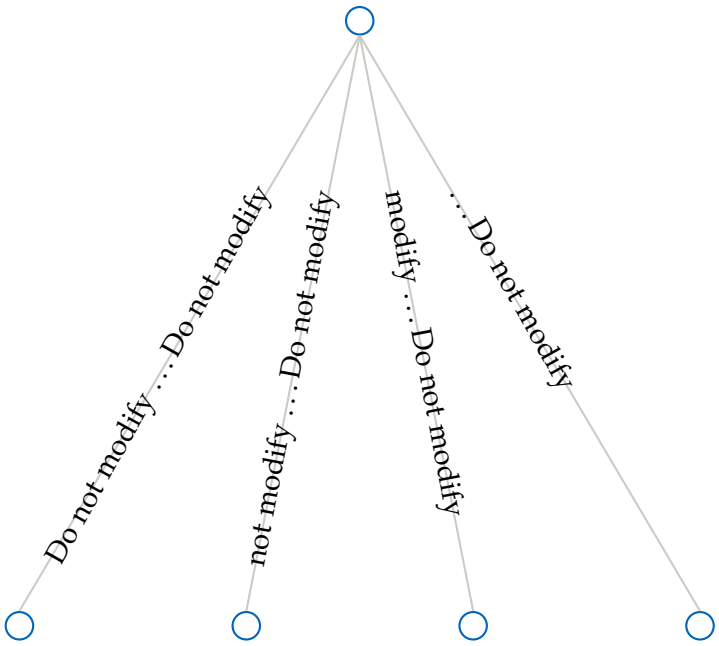


Figure 4.1: The implicit suffix tree for the list of words "Do not modify ... Do not modify".

is an equality function that can be evaluated in linear time to keep the linear time property of the algorithm. Therefore a function has been implemented 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 clone chunks 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

Weiß nicht genau wie ich das erklären soll. To find duplicate sequences in the suffix tree a variety of tree search algorithms can be applied. They are all based on the

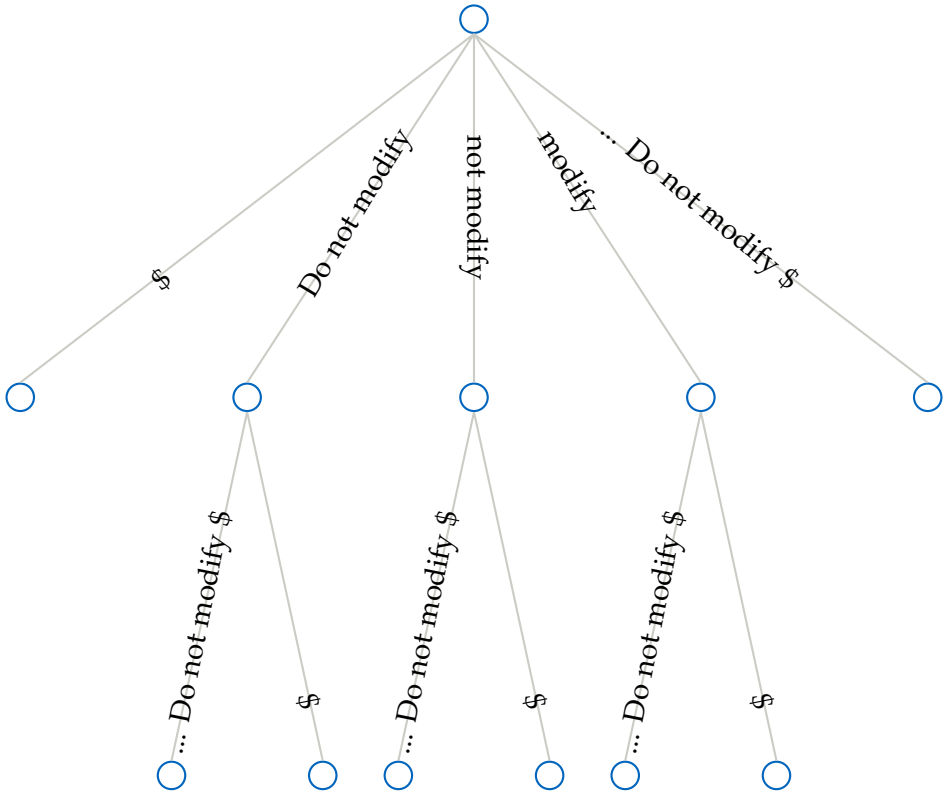


Figure 4.2: The explicit suffix tree for the list of words "Do not modify ... Do not modify".

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.
Im Moment noch einfach kopiert. Dringend umschreiben.

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

```
consider current vertex cur
for (each child of current vertex cur)
       if (P does not match child.label)
              continue
if (a full match) return all results
if (a partial match) go deeper
return no match
```

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 clone chunks. Each inner `List` of the `CloneClass` represents a clone instance. The clones consist of a `List` of clone chunks 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, what will become important in the next steps.

Listing 4.2: The clone classes 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": [
```

```
12        ["The", "following", "code", "was", "generated", "by"],
13        ["The", "following", "code", "was", "generated", "by"],
14        ["The", "following", "code", "was", "generated", "by"]],
15    ]
16 }
```

This `List` of clone chunks 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` and the `line number`.

Listing 4.3: Clone results after processing of the clone classes.

```
1  {
2    "name": "Do␣not␣modify", "occurrences": [
3      [originID: "/projectX/class1.java", lineNumber: "9"]
4      [originID: "/projectX/class2.java", lineNumber: "9"]
5      [originID: "/projectX/class3.java", lineNumber: "9"]],
6    "name": "This␣class␣is␣generated␣by", "occurrences": [
7      [originID: "/projectY/class1.java", lineNumber: "1"]
8      [originID: "/projectY/class2.java", lineNumber: "1"]
9      [originID: "/projectY/class3.java", lineNumber: "1"]],
10   "name": "The␣following␣code␣was␣generated␣by", "occurrences": [
11     [originID: "/projectZ/class1.java", lineNumber: "48"]
12     [originID: "/projectZ/class2.java", lineNumber: "48"]
13     [originID: "/projectZ/class3.java", lineNumber: "48"]],
14 }
```

4.6 Filter possibly generated clone results

As one of the last steps the clone results 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 later by hand performed search for generator patterns in the found clones.
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 clone classes.

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

4.7 Generate links to the files

The last step that is performed by the tool written for this thesis is to generate links to the files where the clones originated from. To do so two folders are created, one holding the possibly generated clone classes and one that holds the ones that are unlikely to be generated.
Within these folders are again folders created that are named by the number of occurrences of the including clone classes.
For each clone class is now again a folder created that is named as the clone class, within which links are created to the files where the clone classes originated from and its respective line number.

Listing 4.5: Folder structure that is used to hold the links to the clones.

```
1  results/
2          possiblyGenerated/
3                  possiblyGenerated_5/
4                          Donotmodify/
5                                  -> class1_9
6                                  -> class2_9
7                                  -> class3_9
8                                  ...
9                  possiblyGenerated_10/
10                         Thisclassisgeneratedby/
11                                 -> class1_1
12                                 -> class2_1
13                                 -> class3_1
14                                 ...
15         notGenerated/
```

```
16              notGenerated_7/
17                      Copyrightc20022003/
18                              -> class1_3
19                              -> class2_3
20                              -> class3_3
21                              ...
```

4.8 Generation of a Generator-Pattern Repository

5 Evaluation

6 Future Work

- In Teamscale einbauen

7 Conclusion

List of Figures

4.1 The implicit suffix tree for the list of words "Do not modify ... Do not modify". 13
4.2 The explicit suffix tree for the list of words "Do not modify ... Do not modify". 14

List of Tables

2.1 Clone Pair and Clone Class 4
2.2 Token properties 6

4.1 Comment types 10
4.2 Comment enclosings 11

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.