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

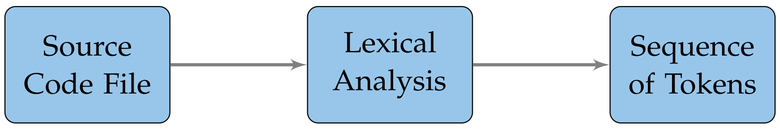
Bessere Überschrift
In this thesis, the approach differs from Bernwiesers so that the clone-detection is done on the comments added to the source code files in the dataset, rather than on the respective source code. A goal of our approach is to find the specific generator patterns that Bernwieser pointed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

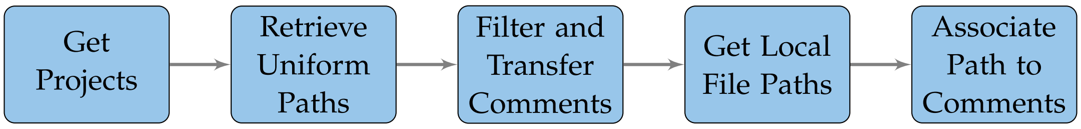


4.2 Connect to the Teamscale server and retrieve comments

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 5.4.1.

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].

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.
This process is highly multithreaded to speed it up. For benchmarks see 5.4.1.

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 $T^i = t_1 \ldots t_i$ can be obtained from the suffixes of string $T^{i-1} = t_1 \ldots t_{i-1}$ 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

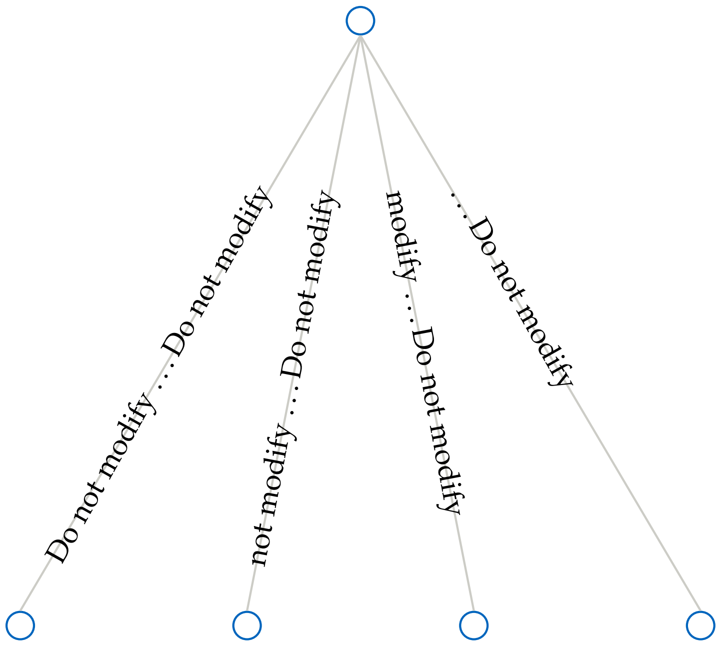

Do not modify . . . Do not modify
not modify . . . Do not modify
modify . . . Do not modify
. . . Do not modify

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 `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 `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 5.4.1.

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)
```

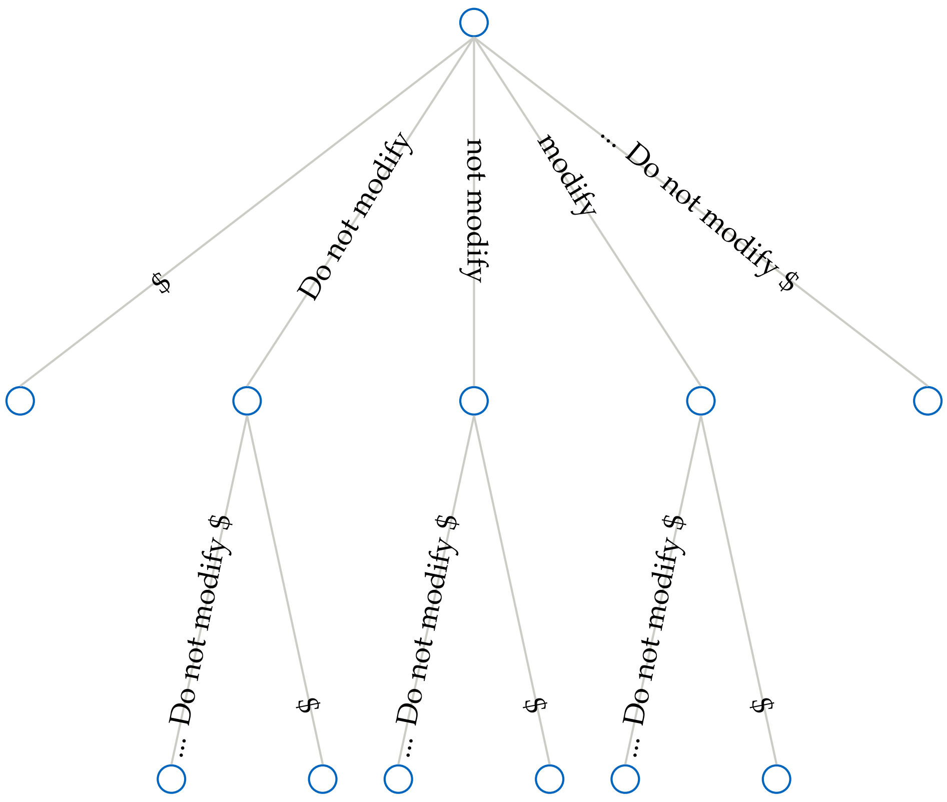


Figure 4.2: The explicit suffix tree for the list of words "Do not modify ... Do not modify".

```
        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 Filter possibly generated CloneResults

In this steps the `CloneResult`s 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 `CloneClass`es.

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

4.7 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 `CloneResult`s 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 `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.

```
{
        "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 `CloneResult`s 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"}]
}
```

4.8 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

   For each generator pattern a folder is created that contains links to all files that include the pattern.

Save as files and as links is highly multithreaded to speed it up. For benchmarks see 5.4.1.

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. The created links and files are viewed and checked whether the proposed generator pattern is indeed an indicator for the file being generated.

Table 4.3: An excerpt of the Generator-Pattern Repository

Generator	Regular Expression Pattern
Apache Cayenne	Class <Class>was generated by Cayenne. It is probably a good idea to avoid changing this class manually, since it may be overwritten next time code is regenerated. If you need to make any customizations, please use subclass.
Apache Thrift	Autogenerated by Thrift ↩DO NOT EDIT UNLESS YOU ARE SURE THAT YOU KNOW WHAT YOU ARE DOING
Compiere	Generated Model - DO NOT CHANGE
Java Annotation	@generated
JavaCC	Generated By:JavaCC: Do not edit this line.
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>
Jena schemagen	Vocabulary Class generated by Jena vocabulary generator
JFlex	NOTE: This class was automatically generated. DO NOT MODIFY.
Jflex	The following code was generated by JFlex <Version>on <Timestamp>
JFlex	This class is a scanner generated by <a href="http://www.jflex.de/">JFlex</a> <Version>on TIMESTAMP from the specification file
SableCC	This file was generated by SableCC (http://www.sablecc.org/).
schemagen	@author Auto-generated by schemagen on <Timestamp>
Snowball	This (class|file) was automatically generated by (a|the) Snowball to Java compiler (↩It implements the stemming algorithm defined by a snowball script.)?

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.
With this data set it is tested how many generator patterns can be found during execution and how many clone classes are falsely classified as not generated.
Why is task relevant

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.
Why is task relevant

5.2 Study Objects

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], the same two subsets of projects have been reused. The classification of the projects included in the two sets has been made in [4], whereas one consists of projects including generated code and the other fully excluding this

Table 5.1: The test environments used for evaluation. [4]

Projects with generated Code	**LOC**	**Projects without generated Code**	**LOC**
Axion	41.862	AOI	153.186
Cayenne	41.862	Aspectj	598.485
Cobertura	68.154	Azureus	831.582
Compiere	727.702	Checkstyle	90.5073
Derby	1.208.453	Collections	109.415
Exoportal	146.947	Ganttproject	69.322
Findbugs	185.912	Hsqldb	269.978
GT2	1.540.009	Htmlunit	174.415
Hadoop	1.064.339	Informa	29.587
Hibernate	897.820	Itext	145.118
Ireport	338.819	JavaCC	35.145
Jena	635.676	Jchempaint	372.743
JHotdraw	133.830	Jext	100.210
Jrefactory	301.940	Jfreechart	313.268
Jstock	74.361	Jgroups	137.614
Lucene	643.243	Jtopen	645.715
PMD	80.971	Maven	111.581
SableCC	35.388	Openjms	111.837
Tomcat	352.572	Poi	363.487
Xalan	354.578	Xerces	237.555

type of source code.
Both collections show a very wide variation range regarding their project sizes; projects with generated code range from 35.388 *(SableCC)* to 1.540.009 *(GT2)*, projects without generated code from 29.587 *(Informa)* to 645.715 *(Jtopen)* [4]. The project *Mahout* has been replaced by *Xalan* because it isn't included in the Qualitas Corpus anymore.
The distribution of the projects over the environments is shown in Table 5.1.

5.2.2 Random Git Projects

Wenns sein muss hinzufügen

5.3 Study Design

This section describes how the study, using the information from the study objects, attempts to answer the research questions. Different proportions of generated and manually maintained code have been calculated to be comparable to the set of all source code S.

5.3.1 Set of generated code G

To evaluate the accuracy and completeness of the generator pattern repository the set of generated code G will get calculated for the used projects.
Furthermore the set of generated code will be split up into two subsets:

- Set of detected generated code DG

  This set is easily derivable by applying the found generator patterns onto the source code files and extracting the ones including the patterns. The proportion of correct as generated classified code DG^+ in contrast to the falsely classified code DG^- is evaluated.

- Set of undetected generated code NG

  From the left set of unclassified source code a manual classification has been performed to extract the source code sections that are obviously generated. At this point we are partially relying on the previously performed classification done in [4] to speed up the process and for a better comparability.

5.3.2 Set of manually maintained code M

After the extraction of generated code the left set of source code is labeled as manually maintained due to the assumption of being mutually exclusive to each other.
Additionally, the set of manually maintained code will again be split up into two subsets:

- Set of detected manually maintained code DM

  This subset can be easily derived by subtracting the set DG^+ from S using the mutual exclusivity.

- Set of undetected manually maintained code NM

  This set will be extracted by investigating into the set DG an searching for wrongly classified manually maintained code that is labeled as generated.

5.3.3 Ratio of generated to manually maintained code

The ratio of generated to manually maintained code $|DG^+|/|DM|$ is calculated to estimate the significance of generated code for software projects.

5.3.4 Ratio of not detected generated code to detected generated code

To evaluate the completeness and accuracy the ratio of not detected generated code to detected generated code $|DG|/|NG|$ is calculated.

5.3.5 Ratio of not detected generated code to detected manually maintained code

5.4 Procedure

5.4.1 Benchmarking

5.4.2 Thresholds

5.5 Results

5.5.1 Qualitas Corpus

5.5.2 Random Git Projects

5.6 Discussion

5.6.1 Completeness and accuracy

5.6.2 Relevance of results

5.7 Threats to validity

5.7.1 Wrong filtering

5.7.2 Minimum clone length vs. irrelevant data

5.7.3 Representativeness of data sets

5.7.4 Generators without pattern

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] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, “The Qualitas Corpus: A curated collection of Java code for empirical studies,” *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, pp. 336–345, 2010, ISSN: 15301362. DOI: `10.1109/APSEC.2010.46`.