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 Clone class 3
2.1.8 Clone result 4
2.2 Metrics 4
2.2.1 Lines of code 4
2.2.2 Clone coverage 4
2.2.3 Comment ratio 4
2.3 Teamscale 4
2.3.1 Suffix Tree Clone Detection 4
2.3.2 Lexer 5

3 Related Work **6**
3.1 Automatic Categorization of Source Code in Open-Source Software - Jonathan Bernwieser 6
3.1.1 Number of clones per clone class 6
3.1.2 Clone coverage per class 7
3.1.3 Results 7
3.1.4 My extension 7

4 Approach **8**
4.1 Use Teamscale as lexer to extract Tokens 8
4.2 Filter tokens to extract comments 8
4.3 Normalize comments for suffix tree clone detection 8
4.4 Build suffix tree 8
4.5 Find clones 8
4.6 Filter possibly generated clone results 8
4.7 Generate links to the files 8
4.8 Generation of a Generator-Pattern Repository 8

5 Evaluation **9**

6 Future Work **10**

7 Conclusion **11**

List of Figures **12**

List of Tables **13**

Bibliography **14**

1 Introduction

Source code of software can be categorized regarding to its role in the maintainability of a software system. The biggest categories besides manually produced 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 source 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 following steps:

- 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 heuristics that detect generated code of several code generators will be derived by means of the comments. Therefore a generator pattern repository will be created.
- The completeness and accuracy of the developed heuristics will be evaluated on a reference data set.
- Automatic classification of source code in a variety of open source systems and evaluation of the amount of generated code.

2 Terms and Definitions

2.1 Terminology

2.1.1 Generated code

— Copied —
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

— Copied —
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 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 generator patterns 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 the target of this thesis. It will be 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 origin. 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 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: i) $< F1(b), F2(b), F3(a) >$, where the three code portions $F1(b), F2(b)$ and $F3(a)$ form clone pairs with each other, ii) $< F1(a), F2(a) >$, and iii) $< F2(c), F3(c) >$ [4].

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 ...*/

2.1.8 Clone result

2.2 Metrics

2.2.1 Lines of code

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

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.
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 second important aspect that is included in Teamscale are the lexers for a variety of programming languages. 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.

3 Related Work

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

Johnathan Bernwieser introduced in his thesis an attempt of automating the process of categorization of source code in [4], whereas he classified the 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.
Following he approached finding generated code 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. Problematic in this approach 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

Based on these results the approach used in this thesis 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. By using this approach the target 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 pattern 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

The research question studied in this thesis is the 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 heuristics that detect generated code of several code generators will be derived by means of the comments. Therefore a generator pattern repository will be 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.

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. These

4.2 Filter tokens to extract comments

4.3 Normalize comments for suffix tree clone detection

4.4 Build suffix tree

4.5 Find clones

4.6 Filter possibly generated clone results

4.7 Generate links to the files

4.8 Generation of a Generator-Pattern Repository

5 Evaluation

6 Future Work

- In Teamscale einbauen

7 Conclusion

List of Figures

List of Tables

2.1 Clone Pair and Clone Class 3

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`.