

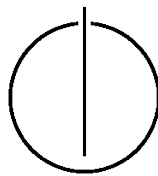
FAKULTÄT FÜR INFORMATIK

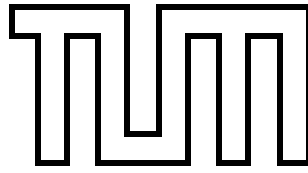
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

# **Automatic Categorization of Source Code in Open-Source Software**

Jonathan Bernwieser





FAKULTÄT FÜR INFORMATIK

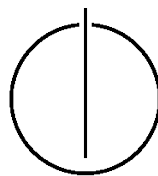
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Automatic Categorization of Source Code in  
Open-Source Software

Automatische Kategorisierung von Quelltext in  
Open-Source Software

Author: Jonathan Bernwieser  
Supervisor: Prof. Dr. Dr. h.c. Manfred Broy  
Advisor: Dr. Elmar Jürgens  
Date: February 11, 2014



I assure the single handed composition of this bachelor's thesis only supported by the declared resources.

München, February 14, 2014

Jonathan Bernwieser

---

## Abstract

The most common method to produce significant information about the quality of a software is the use of different quality metrics. These metrics are based on static analyses which evaluate source code using certain quality patterns. In order to achieve valid results out of such analyses, the relevance of the examined source code is crucial. When measuring maintainability, for instance, miss-categorizing generated and manually-maintained code can lead to results that represent an erroneous image of the real state of a system's maintainability.

In this paper, we investigate the categorization of source code in open-source software. We introduce an attempt of automating the process of categorization and present an empirical study about the quality differences among the categories. Source code is classified in productive and test code, followed by a sub-categorization in "manually maintained" and "automatically generated". Thereby, we introduce a new approach of classifying generated code using clone-detection. By analyzing 110 Java classes, we found evidence for substantial differences of several quality aspects within the code categories. We illustrate that in average 70% of a system's source code consists of manually-maintained production code, making this the most influential category among all others. However, we provide evidence for projects containing over 70% of generated code respectively over 50% of test code. Using these values, we illustrate the importance of categorizing source code in order to produce significant information about the quality of a software. Finally, we conclude our assembled results with directions for further research.

# Contents

<b>Abstract</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Fundamentals</b>	<b>3</b>
2.1 Terminology . . . . .	3
2.2 Classification of Code Categories . . . . .	4
2.3 Metrics . . . . .	5
2.4 ConQAT . . . . .	5
<b>3 Identification of Code Categories</b>	<b>8</b>
3.1 Underlying Architecture . . . . .	8
3.2 Filtering Code Categories Based on Heuristics . . . . .	10
3.2.1 Identification of test code . . . . .	10
3.2.2 Identification of generated code . . . . .	10
3.3 Result . . . . .	11
<b>4 Identifying code generators</b>	<b>13</b>
4.1 Research Questions . . . . .	13
4.2 Objects . . . . .	14
4.3 Design . . . . .	15
4.4 Procedure . . . . .	15
4.5 Result . . . . .	17
4.6 Discussion . . . . .	20
<b>5 Analyzing quality metrics</b>	<b>22</b>
5.1 Research Questions . . . . .	22
5.2 Objects . . . . .	22
5.3 Design . . . . .	24
5.4 Procedure . . . . .	24
5.5 Result . . . . .	25
5.6 Discussion . . . . .	34
5.7 Threats to validity . . . . .	36
<b>6 Related Work</b>	<b>38</b>
6.1 Code Categorization . . . . .	38
6.2 Cloning and clone detection . . . . .	39
<b>7 Future Work &amp; Conclusion</b>	<b>41</b>

**Bibliography**

**43**

# 1 Introduction

Obtaining the standard of a software's quality requires continuous maintenance and control instances. As S. Kan describes in his book "Metrics and models in software quality engineering" [8], quality itself isn't a single idea, but rather a multidimensional concept. It includes the entity of interest, the viewpoint on that entity, and the quality attributes of that entity. This means when people talk about quality, one could be referring to it in a totally different way as another. To capture this large variety of different aspects, quality metrics are the most common utility. With their help, we measure values that give us an idea of certain quality criteria of a software.

Functionality, usability, reliability, performance or maintainability are examples of different parameters that are used in order to measure quality. Thereby, the key to valid results of such quality analyses is the definition of the right measurement scope. Failing this task might lead to distortions within the analysis before it has even started. For instance, test coverage is being used in order to reduce the appearance of software bugs. This analysis executes the existing test suite and marks every productive part of the software as covered which has been touched by at least one of the executed tests. Therefore, to perform a test coverage analysis, it is necessary to distinguish between production and test code within a system. An incorrect definition of the different scopes will inevitably lead to variations in the reported coverage: recognizing test as production code may result in a lower coverage whereas recognizing production as test code is likely to result in a higher coverage.

Another important factor for significant quality analyses is the relevance of the measured source code. For instance, as there are no manually performed modifications within automatically generated code, its appearance does not affect the maintainability of a software. Hence, including generated code in this analysis might create results that do not represent the actual maintainability of the system.

To which degree a result might be distorted, can be seen in table 1.1. It shows the clone coverage within the open-source project "Cobertura" and the percentage shares of LOC of the different code categories. Without considering different code categories, project "Cober-

	LOC	clone coverage
all code	68154 (100%)	83.1%
productive code	12945 (18,99%)	15.9%
generated code	50113 (73.53%)	98.1%
test code	5096 (7.48%)	22.4%

Table 1.1: number (and percentage) of LOC and clone coverage per code category of project "Columba"

tura" provides a clone coverage of 83,1%. This results from the percentage shares of each category. Generated code holds 73,53% of the entire source code and a clone coverage of

98,1%. On the contrary, productive code with a small percentage of 15,9% of LOC, has 18,99% clone coverage.

These extreme differences lead to the following questions: i) which categories should be distinguished in a software analysis, ii) how is the source code distributed within these categories, iii) what are the quality differences and how big is the impact of each category on the overall quality of a software.

### Thesis Outline

To answer these questions, this thesis is organized as follows:

**Chapter 2** summarizes the fundamentals needed to understand this thesis. It presents the different code categories and the metrics used to analyze and distinguish them. Furthermore we give a brief introduction to ConQAT, the tool used to perform the analyses.

**Chapter 3** contains the depiction of the heuristics used to identify and filter each code category. We illustrate the used architecture and the efficiency of the developed identification approaches.

**Chapter 4** describes a case study performed during this thesis in order to identify code generators. We present the technique used and the test environment the analyses were performed on, followed by the description of the analysis process and its outcoming results.

**Chapter 5** is the report of the case study about the analysis of different quality aspects of each code category. We depict the results from the investigated metrics described in chapter 2 and evaluate the impact of each code category.

**Chapter 6** contains work related to code categorization and the quality differences among them.

**Chapter 7** first concludes and summarizes this thesis, followed by an outlook for future investigation steps to optimize the categorization process and extend the knowledge about it.



## 2 Fundamentals

This chapter summarizes the fundamentals needed for understanding this thesis. We will first introduce some technical terms and describe the different categories the tested source code will be subdivided in. Section 2.3 describes what quality metrics were applied in this study, followed by a presentation of ConQAT, the toolkit every performed analysis is built upon.

### 2.1 Terminology

**Clone** A clone refers to a code fragment that is identical or similar to another code fragment. It consists of at least two instances but is not limited to it. The term similarity states that copied code with minor modifications is still considered a clone. The original and the duplicated fragment build a clone pair [2]. Although clones exist not only within source code but also in comments or documentation, in this thesis we define its appearance in source code only.

**Clone class** A clone class is the maximal set of code fragments in which any two of the code fragments form a clone pair. Figure 2.1 depicts an example of the appearance of 3 clone classes: i)  $\langle F1(b), F2(b), F3(a) \rangle$ , where the three code portions F1(b), F2(b) and F3(a) form clone pairs with each other, ii)  $\langle F1(a), F2(a) \rangle$ , and iii)  $\langle F2(c), F3(c) \rangle$  [2]. A clone class consists therefore of a bundle of code fragments with the same clone pattern.

<u>Fragment 1:</u>	<u>Fragment 2:</u>	<u>Fragment 3:</u>
...	...	
<pre>for (int i=1; i&lt;n; i++) {     sum = sum + i; }</pre>	<pre>for (int i=1; i&lt;n; i++) {     sum = sum + i; }</pre>	...
<pre>if (sum &lt; 0 ) {     sum = n - sum; }</pre>	<pre>if (sum &lt; 0 ) {     sum = n - sum; }</pre>	<pre>if (result &lt; 0 ) {     result = m - result; }</pre>
...	<pre>while ( sum &lt; n ) {     sum = n / sum ; }</pre>	<pre>while (result &lt; m ) {     result = m / result }</pre>
	...	...

Figure 2.1: clone classes

**Clone report** XML file created by the clone detection tool of ConQAT. It contains the

output of a successful clone analysis, amongst others the length and number of the containing clones.

### 2.2 Classification of Code Categories

In this section we describe the determined categories in order to classify source code. We based the classification on T. Alves study "Categories of Source Code in Industrial Systems" [1] with slight differences. Although we share T. Alves main categories "productive" and "test", we only subdivide them in "manually maintained" and "automatically generated". The classification of source code is illustrated in figure 2.2. These 4 sub-groups are

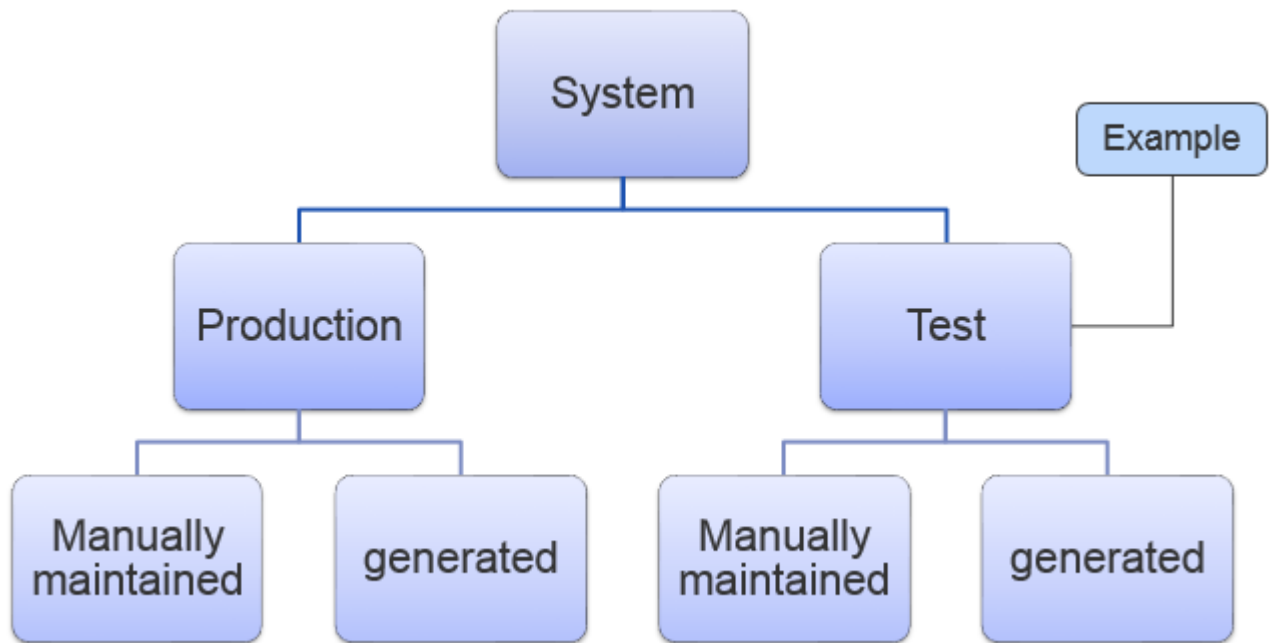


Figure 2.2: classification of source codes

mutually exclusive (same artifact can only be classified in one category) and collectively exhaustive (all categories together cover the entire source code of a system).

**production code:** we consider production code all artifacts that are directly related with the logic of a software product and are necessary to its compilation and execution.

**test code:** there is a different criteria whereby source code is classified as test code. First of all, we consider all artifacts as test code whose purpose it is to invoke functionality of production code for testing purposes. Unlike T. Alves, we then added all example code to this category, whose solely purpose is of demonstrative respectively descriptive nature.

**manually-maintained code:** we classify a code artifact as manually-maintained, if created and modified manually by a developer. There are no restrictions to the use of help

tools (e.g. editors) as long as the resulting source code is produced by humans.

**generated code:** we consider generated code all Java classes that have been created by code generators. Within this context, a code generator is a tool which transforms inserted artifacts automatically into source code (e.g. parsers). We do not distinguish between manually written and generated code within the same Java class. Once a generated code fragment is found, the entire class is considered generated as ConQAT uses file-level granularity to categorize files.

## 2.3 Metrics

This chapter describes the metrics used to determine the quality of each category in relation to the overall quality. Metrics are used to reflect the complexity and other non-functional requirements of a software.

**LOC:** the "Lines of Code" metric represents the size of a software and is thus one of the easiest ways to represent its complexity [5]. 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.

**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 [7]. If the clone rate is too high, it can be very dangerous to implement changes as they have to be performed on every clone.

**comment ratio:** another quality metric used in this study is comment ratio. It is calculated as the ratio of comment lines and entire lines of code. Low comment ratio can lead to decreased maintainability and readability [6].

**method length:** long methods can lead to an increased complexity and a worse readability of a software. They also result in more memory footprint as they are directly proportional to the program memory. The long method metric counts the number of non-blank and non-comment lines inside the method body [9].

**nested depth:** another option to investigate the complexity of software is the "nested depth" metric. It reflects the number of decisions in a control flow that are necessary to perform an action. Figure 2.3 shows a theoretic example of a bad structured condition block. The depth of nested blocks in code can lead to an increasing complexity and a worse readability [9].

## 2.4 ConQAT

This section provides an introduction in the open source toolkit ConQAT as every analysis performed in this thesis is built upon it. ConQAT is developed and maintained by

```

if true then
  if true then
    if true then
      if true then
        if true then
          if true then
            if true then
              if true then
                if true then
                  if true then
                    DeepNesting()
else ()

```

Figure 2.3: example of deep nesting

the CQSE GmbH at the Technische Universität München. A detailed description can be found in the referenced manual "ConQAT Book" written by F. Deißeböck et. al. [3]. It provides tools for rapid development and execution of software quality analysis. In order to give a high level quality overview, dashboards are used to aggregate and visualize data generated during quality analyses. These analyses are featured by a set of interactive tools that support the in-depth inspection of identified quality defects and help to prevent the introduction of further deficiencies.

All operations performed by ConQAT are based on so-called processors. Processors are

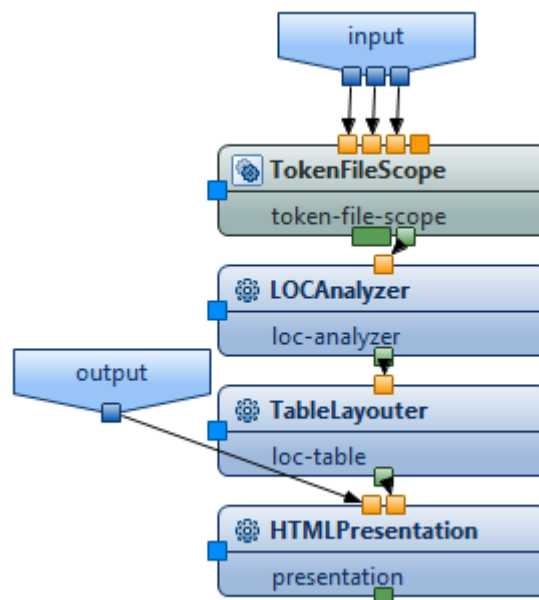


Figure 2.4: LOC analysis implementation of ConQAT

the main execution unit and are defined by a single Java class. Every analysis infrastruc-

ture is built upon the connection of several processors which all perform single tasks [11]. Figure 2.4 shows the implementation of the LOC analysis. Firstly the *"TokenFileScope"* reads a text file on token level into the system. Then, the *"LOCAnalyzer"* calculates the LOC of every file and stores this value in an hash table; this information is forwarded to the *"TableLayouter"* and the *"HTMLPresentation"* processor which build an HTML dashboard to provide a high-level quality overview.

All analyses performed in this thesis are based on particular compositions of the respective analyzer processors and the advanced clone detection tool provided by ConQAT. Although we reduced our language scope to Java only, ConQAT provides different options to investigate simple text files with the extention to specify on various programming languages. The creation of individual processor architectures is based on a graphical Eclipse integration.

## 3 Identification of Code Categories

Generally, there exist no naming conventions which make source code assignable to a certain code category. Thus, it is necessary to investigate different identification approaches which classify code as optimal as possible. This is done via heuristics. In this chapter we describe the development of these heuristics used to identify and filter each code category. We illustrate the used architecture and illustrate the efficiency of our filtering approaches.

### 3.1 Underlying Architecture

In order to identify different code categories, an existing benchmark of both generated code as well as test code is needed to be able to develop filter heuristics. Based on these benchmarks, different identification approaches can be investigated and optimized. To filter all 4 code categories defined in chapter 2, it is sufficient to build one architecture to separate test code and one architecture to separate generated code. By combining these filtering mechanisms as follows, it is possible to refine the resulting Java classes to "*manually-maintained*" and "*generated*":

1. *productive manually-maintained*:  $\neg \text{test code} \wedge \neg \text{generated code}$
2. *productive generated*:  $\neg \text{test code} \wedge \text{generated code}$
3. *test manually-maintained*:  $\text{test code} \wedge \neg \text{generated code}$
4. *test generated*:  $\text{test code} \wedge \text{generated code}$

The architectures used to run these investigations are built and executed with ConQAT. The creation of the Benchmark was performed manually by examining the systems used in our study.

To limit the search scope, we defined certain basic criteria a Java class must in any case correspond to, to be part of one of the two code categories: test classes must either contain the string "*test*" (or "*sample*") or be filed in a folder containing these strings. Generated classes must either contain the string "*generated*" or be filed in a folder containing this string.

The creation of the benchmark for test code was based on a simple filtering mechanism performed with ConQAT. Thereby every Java class was marked as "true" if it applied to the respective patterns mentioned above. The verification whether or not the particular Java class was in fact for testing purposes, was done by checking randomly the benchmark of matching files manually. The resulting collection has a total of 26.820 test classes. The same procedure was used for generated code. In order to complete this collection as best possible, we consulted the development teams of every project personally for any disregarded classes which led to a total amount of 3.030 generated files.

The main ConQAT architectures used to develop the filter heuristics for both code categories are depicted in Figure 3.1.

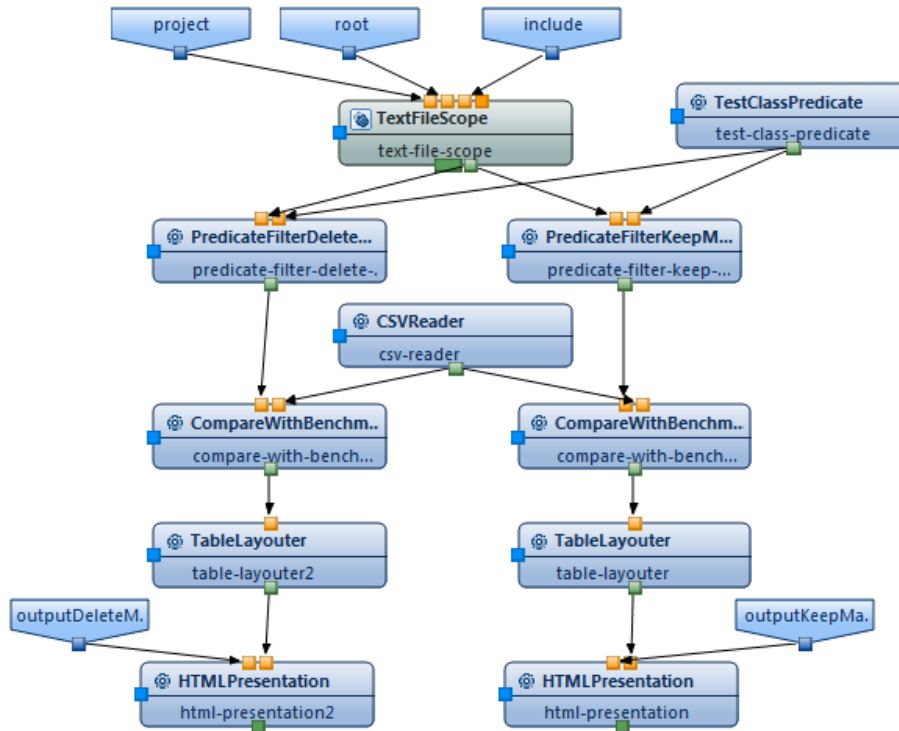


Figure 3.1: ConQAT implementation of the filtering process

After being inserted into ConQAT by the *TextFileScope* processor, the project bundle runs through two kinds of filter processors; both, *PredicateFilterDeleteMatch* as well as *PredicateFilterKeepMatch*, are needed to verify the efficiency of the actual heuristic. The *PredicateFilterKeepMatch* processor keeps only these classes matching the search query. Thus, at best case, only these files are forwarded which provide the testing purpose (respectively, are generated). To verify the correctness of this filtering, the *CompareWithBenchmark* processor compares the remaining classes with the respective benchmark. If a forwarded Java class is found in the benchmark, it will be marked as *true*, not found classes as *false*. *False*-branded files are classes that are mistakenly recognized by the heuristic as one of the respective code categories and thus are false positives.

On the other hand, the *PredicateFilterDeleteMatch* processor eliminates all classes matching the search query; hence, every forwarded class should not correspond to the respective code category and thus be tagged with *false* by the *CompareWithBenchmark* processor. Files that are marked with *true* did not get recognized by the heuristic and hence are false negatives.

Both filter processors use the *TestClassPredicate* processor (*GeneratedClassPredicate* respectively for the identification of generated code) which is the processor holding the actual deciding mechanism of the heuristic.

## 3.2 Filtering Code Categories Based on Heuristics

The correct degree of specification of a heuristic decides the validity of the resulting outcome. Defining too specific patterns to search might distort the result as it raises the likelihood of excluding many classes that are part of the respective code category. For instance, searching the string "test class of project *exampleproject*" is sure to select only classes that do contain test code. Still, it would only identify Java classes within this specific project and ignore the rest.

On the other hand, extreme generalization of the search patterns turns the final results vice versa; reducing the search string to the word "test" cannot guarantee the identification of only test classes as there are many possibilities a single word is used in a different context. Hence, in order to optimize our heuristics, we ran several test runs starting with very generalized search patterns ("test", "generated") towards a degree of specification that seemed to produce reasonable results. The specification process was done manually by analyzing the outcome of every test run and comparing it with the respective benchmark.

The following describes the developed heuristics to identify test and generated source code. The case-sensitivity has to be taken into account as it has a major influence on the resulting outcome:

### 3.2.1 Identification of test code

The way of localizing test code is divided into three different approaches;

1. *The class is filed in a "test"-directory:* Every class that is saved within a folder containing the word "test", "Test", "sample", "Sample", "example", "Example" in its name is classified as test code.
2. *The class is named a test class:* Every class containing the word "Test", "Example", "Sample" in its name is recognized as test class. As the Java programming guidelines recommend class names to be capitalized, case-sensitivity reduces the likelihood of false positives.
3. *The class inherits from a test class:* Every class that extends a test class is also classified a test class. The identification is hereby done using regular expression.

### 3.2.2 Identification of generated code

The localization of generated code turned out to be more complex than the identification of test code. This relies on several reasons; generated classes normally do not identify themselves by neither their naming nor their inheritance; however, it appears more problematic that there is no guidelines about the development of code generators which normalize the way generated code is identifiable.

Although it is a very common feature of code generators that their created source code is commented with information containing the substring "generated by", it must not be seen as a general convention. Instead, "generated by" is a popular word combination used by many developers as part of their manual written comments. Reducing the search pattern to it would hence produce a high rate of false positives. Therefore, as there are no trustworthy commonalities within generated code that grant an adequate degree of correctness, we



decided to maximize the specification level of our heuristic using clone detection. Thereby

```
"This class was generated by the JAXRPC SI, do not edit"
"Generated file - Do not edit!"
"Automatically generated by JJTree"
"Auto-generated by schemagen"
"Auto-generated by jena.schemagen"
"Vocabulary Class generated by Jena vocabulary generator"
"AUTOGENERATED FILE: DO NOT EDIT"
"The java class is generated from hql-sql.g by ANTLR"
"DO NOT EDIT THE GENERATED JAVA SOURCE CODE"
"This code generated using Refrations SchemaCodeGenerator"
"The following code was generated by JFlex"
```

Figure 3.2: strings created by code generators

we performed an external analysis to identify the code generators used and how they tag their generated code. Every identification mark of a generator that was found was saved externally and used for the localization of generated code in the categorization process. The exact procedure of the identification of code generators is described in chapter 4. A small extract of the resulting identification marks of code generators is depicted in figure 3.2.

Like for the identification of test code; we also defined every class filed within a folder containing the word *"generated"* in its name as *"automatically generated"*.

### 3.3 Result

To measure the efficiency of the developed heuristics, we performed a precision-recall analysis. The quantitative results of this analysis are depicted in table 3.1. Although there

	test code	generated code
precision	98,65%	99,73%
recall	100%	100%

Table 3.1: efficiency of heuristics

are no official guidelines about writing test code, the efficiency for test code identification shows certain commonalities all test classes seem to share. The precision value states that 1,35% of the classes detected by the heuristics are false positives. This represents only 343 classes. The reason for this miss-interpretation is mainly due to direct implementations of testing tools (e.g. *JUnit*); these tools execute testing operations and hence are often located within directories containing the word *"test"* or provide source code that matches the search patterns but does not belong to the code category *"test code"*.

The recall value of 100% of both identification approaches states that every class within both benchmarks was indeed detected by the heuristics. The precision value for detecting generated code is 99,73%. The missing 0,27% represent 9 Java classes that were mistakenly identified as automatically generated. These results provide evidence for the high

efficiency of the heuristics and thus its sufficiency to be used for further categorization analyses.

## 4 Identifying code generators

In order to filter automatically generated source code, it is essential to identify certain criteria that distinguish generated from manually created code. One cannot rely on a search algorithm that scans Java files for the string "generated" as there exist many possibilities this word can be part of manually written source code. One example of such code is depicted in Figure 4.1. It uses the string "generated" as part of an explanation comment. Therefore,

```
* If there are no parameters, then "()" is returned.  
*  
* @param parameterListOwner the 'owner' of the parameters  
* @return the generated parameter list  
*/  
private String generateParameterList(Object parameterListOwner) {  
    Iterator it =
```

Figure 4.1: code fragment containing the string "generated" as return statement

to assure best possible correctness of the filtering process, it is necessary to identify actual code generators and how their produced source code can be distinguished from other code.

This chapter contains the procedure of identifying code generators and the tags they use to mark their generated source code. We describe the process of performing a clone detection analysis and how we gain knowledge from its outcoming results about these generators.

### 4.1 Research Questions

Every code generator follows given patterns or templates to transform certain input into productive source code. Hence, there should be a variety of commonalties that classes, if generated by the same generator, share. Logical consequences could be an increased clone density or an increased clone coverage.

This leads to the following questions:

**RQ1** *Is it possible to detect code generators performing a clone-detection analysis?*

We need to examine whether or not there are differences between the clone density of generated and manually-maintained code. If the clone density within generated code is higher than average, clone classes containing generated code should be one of the largest among all others clone classes. We investigate the approach of using this fact to identify code generators.

**RQ2** *Can the detection of code generators be atomized?*

In order to build an automated method of identifying source code, unique characteristics are needed that distinguish each code category. Thereby, we refer to a potentially higher clone coverage within generated code. We investigate the possibility of building a mechanism that automatically detects generated code by filtering only classes above a certain clone coverage value. Furthermore, we check the size of clone classes for differences, if comprised by generated code.

## 4.2 Objects

In order to measure the effectiveness of our identification approaches we used two different test environments. One contained only systems with generated code and one only systems without any generated code. All projects used are part of the Quality Corpus which is a collection of open-source systems for use in empirical studies created and maintained by Ewan Tempero [12]. How the systems are distributed among the two software groups is depicted in table 4.1: the left column lists projects with generated code, the right column projects with only manually-maintained 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*).

Projects with generated Code	LOC	Projects without generated code	LOC
Axion	41.862	AOI	153.186
Cayenne	341.913	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
Mahout	288.622	Maven	111.581
PMD	80.971	Openjms	111.837
SableCC	35.388	Poi	363.487
Tomcat	352.572	Xerces	237.555

Table 4.1: the two test environments used in this case study

## 4.3 Design

The investigation, whether or not generated code contains a higher clone coverage or a higher clone density, was done using clone detection. Assuming the higher density, clone classes consisting of generated classes should contain the most clone instances and hence occupy the first places in a list of clone classes that is sorted in descending order according to their size. To verify this, we ran the clone detection analysis on every project that knowingly contains generated code and checked their resulting clone classes with the most clone instances for code produced by a generator. Hereby, we excluded test code from the analysis as test code often contains a high clone coverage and therefore leaves generated code undiscovered; we based the exclusion on the fact that automatically generated test code won't affect further steps of this thesis and can therefore be ignored. This exclusion is performed within all analyses of this chapter and will thus not be mentioned again.

In order to find a way to automate the detection of code generators, it is mandatory to find at least one attribute that distinguishes generated from manually-maintained code. As a generator only produces code by following its patterns, its output can be expected to be very similar; therefore, we implemented two mechanisms that examined source code for the following clone metrics:

1. Number of clones per clone class
2. Clone coverage per class

We performed several runs of the clone detection analysis on both project collections using two kind of filters:

- (i) a clone class filter that removes every clone class that does not contain a certain number of clones and
- (ii) a clone coverage filter that removes every Java class that stays below a certain percentage of clone coverage. We thereby exclude every file that does not reach a certain file size (counted in LOC).

This exclusion is based on previous analyses which showed high similarities of small files within the same project. This leads to such clone coverage among them that their value exceeds the clone coverage of clone classes with generated code. Thus, with their elimination, we increased the possibility of finding generated code.

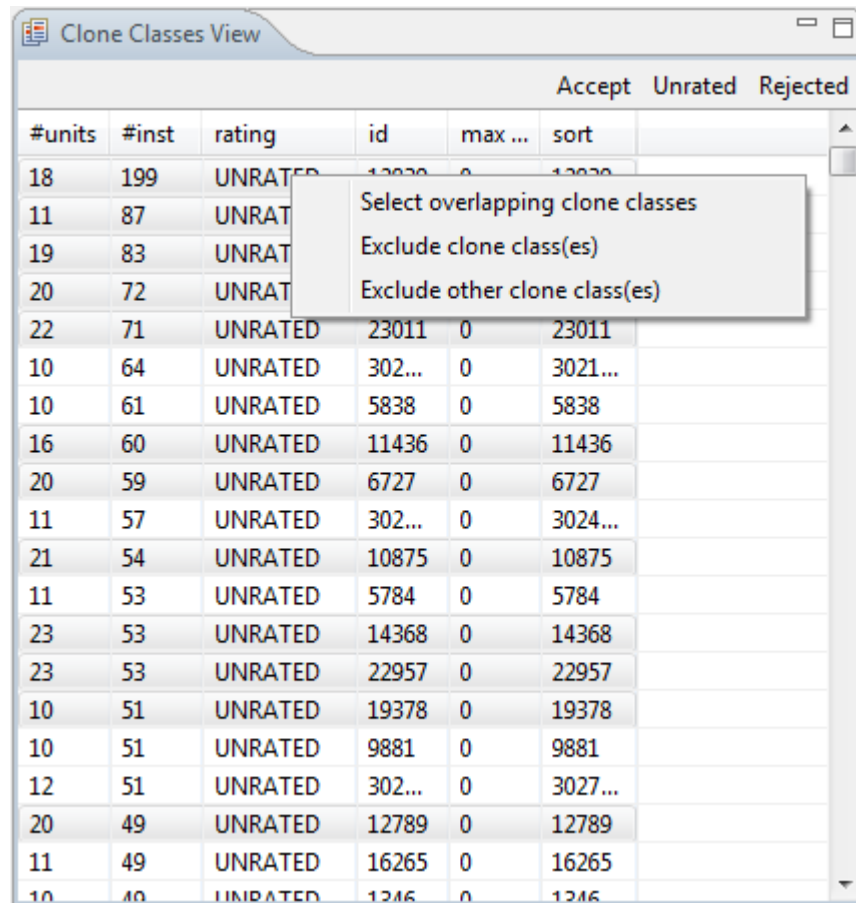
To implement the clone class filter, we searched for clone classes containing more instances than a certain limit after each clone detection analysis; if such a clone class appeared, we manually checked its containing clones for generated code. The elimination of Java files with low clone coverage or size is done automatically by a new implemented filter in ConQAT. While repeating the analysis several times, we adjusted the values of the filters to achieve more accurate results and check for crucial differences between the code categories. Both filters were tested and evaluated separately to provide evidence which filter is responsible in case of success or failure.

## 4.4 Procedure

Before we started the filtering patterns to investigate the possibility of automation, we performed a clone detection analysis without any exclusions on every project of both test

environments to evaluate the efficiency of finding generated code based on clone detection. The clone detection was done using the ConQAT feature "clone detection" with its provided processor "*JavaCloneChain*"; this processor works through given Java files, investigates different kind of clone metrics (e.g. clone coverage, clone classes, number of clones) and stores them in an XML clone report [3]. After the clone-detection process was completed successfully, we manually iterated 4 times through the steps described below to identify code generators using the ConQAT Clone Detection feature:

1. Sort the result list of clone classes in descending order according to their size,
2. Check the Java classes contained within the biggest clone class for generated code
3. Remove the examined clone class from the result list and all clone classes overlapping with it.



#units	#inst	rating	id	max ...	sort
18	199	UNRATED	13030	0	13030
11	87	UNRATED			
19	83	UNRATED			
20	72	UNRATED			
22	71	UNRATED	23011	0	23011
10	64	UNRATED	302...	0	3021...
10	61	UNRATED	5838	0	5838
16	60	UNRATED	11436	0	11436
20	59	UNRATED	6727	0	6727
11	57	UNRATED	302...	0	3024...
21	54	UNRATED	10875	0	10875
11	53	UNRATED	5784	0	5784
23	53	UNRATED	14368	0	14368
23	53	UNRATED	22957	0	22957
10	51	UNRATED	19378	0	19378
10	51	UNRATED	9881	0	9881
12	51	UNRATED	302...	0	3027...
20	49	UNRATED	12789	0	12789
11	49	UNRATED	16265	0	16265
10	40	UNRATED	1246	0	1246

Figure 4.2: excluding overlapping clone classes from sorted list

Figure 4.2 depicts a list of all determined clone classes after a successful clone detection analysis. All classes marked in gray overlap with the first placed clone class and will be removed from the list by selecting *Exclude clone class(es)*.

(i) *Clone class filter*

For our attempt to filter small clone classes it was mandatory to access its size which wasn't yet implemented in ConQAT. Therefore, before processing, we extended the processor *"CloneReportWriter"* which is used by the *"CloneReportWriterProcessor"* to write the clone report XML files. Next to the existing clones.xml we added the feature to create a second report *"cloneswithsize.xml"* which also contains the size of each clone class. The analysis was set to mark every clone class containing more than 20 clones as potentially generated and write it into an external log file.

(ii) *Clone coverage filter*

To exclude Java files with low clone coverage or size from the XML clone report, we implemented a new processor *"CloneCoverageFilter"*. This processor checks consecutively every Java class for its clone coverage and size and removes it if it stays below the limiting parameters.

We ran the analysis 3 times with adjusted values for the minimal clone coverage and minimal number of lines:

1. For the first analysis the filter was parameterized to use 10% as minimal clone coverage and 20 as minimal number of lines.
2. The second analysis was performed with a minimal clone coverage of 20% and 20 as minimal number of lines and
3. the last run was set to a minimal clone coverage of 10% and 50 as minimal number of lines.

To identify generated code we manually scanned through the resulting clone reports by performing the same three steps described above 4 times.

## 4.5 Result

The quantitative results of our clone detection analysis without any previous filtering are depicted in figure 4.3. As described in section 4.4 we performed 4 iterations in our manually performed generator identification approach by consecutively removing the examined clone class and all its overlapping clone classes with it. Every column in both tables represents one iteration; if the identification approach in a certain run was successful, the respective column was marked with an (x).

The left table shows the outcome of the analysis of the system collection containing generated code. Not only was the detection of code generators successful on every project but also 16 out of 20 projects contain their generated Java classes within the biggest clone class which is represented by a successful approach in the first iteration.

In 12 projects were at least 2 clone classes containing generated code; even though this provides evidence for a high clone coverage within generated code, it isn't enough to pronounce it a significant attribute of code generators to be used as a filter. As expected, the right table doesn't contain any entries because there exists no generated code within the second test environment.

(i) *Clone class filter*

Aim of this filtering attempt is the identification of the minimum size of a clone class to make it a potential holder of generated code; in other words, can one be sure, a clone class

Projects	Run 1	Run 2	Run 3	Run 4	Projects	Run 1	Run 2	Run 3	Run 4
Axion			x		AOI				
Cayenne	x		x	x	Aspectj				
Cobertura	x	x	x		Azureus				
Compiere	x				Checkstyle				
Derby	x		x		Collections				
Exoportat		x	x		Ganttproject				
Findbugs	x				Hsqldb				
GT2	x	x		x	Htmlunit				
Hadoop		x	x	x	Informa				
Hibernate	x	x	x	x	Itext				
Ireport			x		JavaCC				
Jena	x	x			Jchempaint				
JHotdraw			x		Jext				
Jrefactory	x				Jfreechart				
Jstock			x		Jgroups				
Lucene	x	x			Jtopen				
Mahout	x	x	x	x	Maven				
PMD	x	x	x	x	Openjms				
SableCC	x	x	x	x	Poi				
Tomcat	x		x		Xerces				

Figure 4.3: detection of generated code with no filter mechanisms

is not comprised of generated code fragments, if it does not reach a certain volume. As described in section 4.4, the minimum size of a clone class to be marked as potentially generated was set to 20.

After the analysis of every project containing generated code, 15 out of 20 projects contained clone classes exceeding this minimum size; in contrast, the category without generated code only provided 6 out of 20 projects. Although this result emphasizes the higher clone coverage within generated code, 5 projects within the test environment with generated code still contain their generated code within smaller clone classes. Therefore, it does not satisfy the approach of finding every code fragment automatically created by a code generator.

To identify the lowest size the clone class filter need to be set to detect all generators, we compared the sizes of all examined clone classes after each iteration of the clone detection analysis without filtering. The results are illustrated in figure 4.4; every cell contains the size of the biggest clone class of a project after each iteration. Clone classes comprised of generated code are specifically highlighted.

Although there is in fact much generated code found in big clone classes (e.g. 664 instances in project *Compiere*), there exist also very small clone classes made of generated code (6 instances in project *Axion* and *Jstock*).

Thus, to identify every code generator in our test environments, 6 needs to be the minimum limit of the clone class filter. This value is far from automatizing the identification of code generators as almost every examined clone class exceeds this limit. Furthermore, it represents the minimum value for the projects used in this case study and cannot be taken



Projects	Run 1	Run 2	Run 3	Run 4
Axion	7	6	6	5
Cayenne	23	14	12	12
Cobertura	60	12	8	4
Compiere	664	447	126	118
Derby	25	17	15	11
Exoportal	15	12	12	8
Findbugs	13	5	4	4
GT2	199	64	51	49
Hadoop	65	42	41	15
Hibernate	116	56	47	21
Ireport	171	90	43	38
Jena	28	22	20	16
JHotdraw	8	8	7	6
Jrefactory	192	19	13	12
Jstock	13	10	6	6
Lucene	37	25	21	20
Mahout	94	72	10	10
PMD	106	20	13	7
SableCC	38	32	19	13
Tomcat	24	10	8	8

Projects	Run 1	Run 2	Run 3	Run 4
AOI	10	10	9	8
Aspectj	62	31	18	17
Azureus	17	15	14	12
Checkstyle	2	2	2	2
Collections	9	6	6	5
Ganttproject	5	5	3	3
Hsqldb	45	11	5	5
Htmlunit	4	4	4	3
Informa	8	4	3	3
Itext	5	4	4	4
JavaCC	10	7	4	4
Jchempaint	74	33	29	29
Jext	10	8	7	5
Jfreechart	50	9	7	6
Jgroups	5	4	3	3
Jtopen	106	27	22	18
Maven	19	8	6	5
Openjms	6	6	4	4
Poi	46	26	24	8
Xerces	18	14	12	10

Figure 4.4: size of biggest clone class after each iteration

as generally valid. The automation of identifying code generators based on a clone class filter is therefore not possible.

(ii) *Clone coverage filter*

The automation of the generator identification using a clone coverage filter aims towards small or in a best case scenario empty XML clone reports of analyzed projects without generated code.

Figure 4.5 shows the results of our first analysis using a minimal clone coverage of 10% and a minimal LOC amount of 20. Every mutation within an iteration step compared to the previous analysis without filtering mechanisms is marked in red: a red (x) represents a newly detected clone class containing generated code, a red marked *project name* an empty XML clone report.

5 projects out of 20 of the test environment without generated code and only one project (*Findbugs*) within the other test environment provide empty XMLs. Although the exclusion of *Findbugs* decreases the possibility of finding every generated fragment using the clone coverage filter, we ran a second analysis increasing the clone coverage to 20% and the LOC limit to 50, as the test environment without generated code held not only 5 empty but also very decreased XMLs after the first analysis. Figure 4.6 confirms the increasing amount of empty clone reports especially for projects without generated code by increasing the value of the clone coverage filter. 13 out of 20 projects without generated code provided empty XMLs and are thus enhancing the automatism of finding generated code. Nevertheless, there were also 7 projects within the test environment with generated code whose generated code was excluded from the analysis by the clone coverage filter.

Projects	Run 1	Run 2	Run 3	Run 4
Axion			x	
Cayenne	x		x	x
Cobertura	x	x	x	x
Compiere	x			
Derby	x	x	x	x
Exoportal	x	x		x
Findbugs				
GT2	x	x		x
Hadoop		x	x	x
Hibernate	x	x	x	x
Ireport			x	
Jena	x	x		
JHotdraw			x	
Jrefactory	x			
Jstock			x	
Lucene	x	x		
Mahout	x	x	x	x
PMD	x	x	x	x
SableCC	x	x	x	x
Tomcat	x		x	

Projects	Run 1	Run 2	Run 3	Run 4
AOI				
Aspectj				
Azureus				
Checkstyle				
Collections				
Ganttproject				
Hsqldb				
Htmlunit				
Informa				
Itext				
JavaCC				
Jchempaint				
Jext				
Jfreechart				
Jgroups				
Jtopen				
Maven				
Openjms				
Poi				
Xerces				

Figure 4.5: detection of generated code with applied clone coverage filter: >10% clone coverage, >20 LOC

By running a final analysis with using the same LOC filter but reducing the clone coverage back to 10% we confirmed the irrelevance of the size of a Java file as there were no differences recognizable compared to the analysis with the clone coverage filter set to 10% (figure 4.5). Thus, although the use of a clone coverage filter seems reasonable as many systems without generated code were automatically excluded, it cannot be used for automating the generator code identification because there were also too many clones consisting of generated code eliminated from the XML reports.

## 4.6 Discussion

In the previous sections we explored the possibility of detecting and automating the detection of generated code using clone detection. In our experiments we determined that every generated code fragment can indeed be identified by iterating through the resulting clone reports at least 3 times. Although every generator was found after the third iteration, which makes the 4th run appear to be redundant, we recommend performing it anyway as there were in fact generated code fragments (created by a generator detected in previous iterations) found in it and it thus increases the possibility of finding more unknown generators. The automation using a clone class filter appeared to be impossible as there was no correlation found between the size of a clone class and the possibility of it consisting of generated code.

In contrast, even though the size of a Java class appeared to be irrelevant, the clone cov-

Projects	Run 1	Run 2	Run 3	Run 4	Projects	Run 1	Run 2	Run 3	Run 4
Axion					AOI				
Cayenne					Aspectj				
Cobertura	x	x	x	x	Azureus				
Compiere	x				Checkstyle				
Derby					Collections				
Exoportal					Ganttproject				
Findbugs					Hsqldb				
GT2	x	x		x	Htmlunit				
Hadoop					Informa				
Hibernate	x	x	x	x	Itext				
Ireport			x		JavaCC				
Jena	x	x			Jchempaint				
JHotdraw			x		Jext				
Jrefactory	x				Jfreechart				
Jstock			x		Jgroups				
Lucene	x	x			Jtopen				
Mahout	x	x	x	x	Maven				
PMD	x	x	x	x	Openjms				
SableCC	x	x	x	x	Poi				
Tomcat					Xerces				

Figure 4.6: detection of generated code with applied clone coverage filter: >20% clone coverage, >50 LOC

erage filter provides evidence for a certain correlation between clone coverage and code generation. After filtering every Java class containing less clone coverage than 20%, 13 out of 20 non-generated projects provided empty clone reports which means there were no clones detected. Although this appears to be a reasonable approach, it cannot be used for a trustworthy generator detection as 7 projects containing generated code weren't found in the analysis either.

It has to be mentioned though, that there exists also a relation between the size of clone classes and the generated code fragments not found after increasing the clone coverage filter: *Compiere*, *GT2* or *Hibernate* with 662, 199 and 116 instances of generated clones within one clone class, for instance, did not eliminate any of their generated code classes whereas *Findbugs*, *JHotdraw* or *Jstock* with 13, 8 and 13 instances excluded them after 10% respectively 20% of clone coverage filtering.

There was no way found though on how to use this correlation to automate the code generator identification with the result of dismissing the automation approach. Nevertheless the detection of every generator worked out successfully which makes it usable for further code categorization heuristics in a semi-automatically way.

## 5 Analyzing quality metrics

In this chapter we analyze the underlying question how different code categories affect the overall quality of software. In order to investigate this, we break the entire source code into the two main code categories "productive" and "test", followed by a sub-categorization into "manually maintained" and "automatically generated". Every sub-category is then analyzed independently to answer the following research questions.

### 5.1 Research Questions

**RQ1** *How much code is in which category?*

How influential a code category is, depends firstly on its proportional share of the project. The bigger a category is, the more it affects the overall quality of a software. This strong correlation between a category's size and its impact on the quality claims the need of identifying each size in order to produce significant results in our further analyses.

**RQ2** *How differ the code categories in different quality metrics?*

Once we have explored the size of each category we compare different quality metrics within and between the code categories. Knowledge about commonalities within the same and differences between them is necessary to refer to the influence of a category over the overall quality of a software. We need to investigate the efficiency of the categorization; some quality metrics might resemble each other through all categories in such manner that distinguishing them isn't profitable after all.

### 5.2 Objects

Unlike the investigations in chapter 3, we extended our test environment to the entire collection of the Qualitas Corpus. The Qualitas Corpus is a curated collection of software systems intended to be used for empirical studies of code artefacts. It is created and maintained by Ewan Tempero and contains a total of 112 Systems [12]. We excluded the projects "Eclipse\_IDE" and "Netbeans" from our investigations as they caused heap space issues during the analysis process. Instead we added project "mahout" to our collection as it marks generated code differently from other projects. All projects used with their respective LOC are listed in Table 5.1. The projects range from small (*Fitjava* 6.991 LOC) to large systems with over 1.000.000 LOC (*GT2* 1.540.009 LOC). In order to optimize the accuracy of our outcoming results, we ran our analyses on every project independently. Thus, it was possible to compare every single analysis with a summarized overall result to verify regularities within each code category.

Project	LOC	Project	LOC	Project	LOC
Ant	256.041	Informa	29.587	Lucene	643.243
Antlr	58.935	Ireport	338.819	Mahout	288.622
AOI	153.186	Itxt	145.118	Marauroa	37.275
Argouml	389.952	Ivatagroupware	71.851	Maven	111.581
Aspectj	598.485	Jag	28.957	Megamek	362.260
Axion	41.862	James	83.716	Mvnforum	172.855
Azureus	831.582	Jasml	7.272	Myfaces_core	189.954
Batik	366.507	Jasperreports	347.502	Nakedobjects	215.016
C.jdbc	174.972	JavaCC	35.145	Nekohtml	13.342
Castor	349.301	Jboss	968.893	Openjms	111.837
Cayenne	341.913	Jchempaint	372.743	Oscache	19.702
Checkstyle	90.5073	Jedit	176.672	Picocontainer	15.999
Cobertura	68.154	Jena	635.676	Pmd	80.971
Collections	109.415	Jext	100.210	Poi	363.487
Colt	84.592	Jfin_DateMath	16.686	Pooka	72.167
Columba	149.498	Jfreechart	313.268	Proguard	101.330
Compiere	727.702	Jgraph	59.145	Quartz	62.229
Derby	1.208.453	Jgraphpad	43.652	Quickserver	30.239
Displaytag	38.729	JgraphT	41.887	Quilt	13.035
Drawswf	46.535	Jgroups	137.614	Roller	135.210
Drjava	160.308	Jmoney	11.304	Rssowl	174.209
Emma	39.676	Joggplayer	51.654	Sablecc	35.388
Exoportat	146.947	Jparse	32.270	Sandmark	128.993
Findbugs	185.912	Jpf	22.521	Springframework	627.186
Fitjava	6.991	Jrat	31.084	Squirrel_sql	9.082
Fitlibraryforfitnesse	42.233	Junit	13.431	Struts	261.773
Freecol	205.085	Jhotdraw	133.830	Sunflow	27.408
Freeecs	29.943	Jmeter	182.575	Tapestry	182.151
Freemind	86.244	Jrefactory	301.940	Tomcat	352.572
Galleon	135.442	Jruby	244.774	Trove	9.768
Ganttproject	69.322	Jspwiki	110.005	Velocity	70.804
GT2	1.540.009	Jstock	74.361	Wct	99.622
Hadoop	1.064.339	JsXe	35.307	Webmail	18.074
Heritrix	126.652	Jtopen	645.715	Weka	496.737
Hibernate	897.820	Jung	67.024	Xalan	354.578
Hsqldb	269.978	Jung	67.024	Xmojo	43.249
HtmlUnit	174.415	Log4j	68.612		

Table 5.1: projects of the Qualitas Corpus

### 5.3 Design

In order to answer our research questions, we used the following study design. As we extended our study objects to the entire Quality Corpus, we needed to extend our collection of generator tags as well; therefore, we ran the clone detection analysis described in chapter 4 on these projects of the Corpus that had not been scanned for generators in our previous analyses. Instead of running the clone detection on the entire rest of the Qualitas Corpus though, we first applied the filter for generated code with the generator tags known so far; this was done for the following purpose: i) reducing the amount of Java classes to optimize runtime ii) investigate the efficiency of filtering generator tags.

After completing the collection of generator tags we used our filter heuristics verified in chapter 3 to isolate each code category and start our analysis. Every analysis was also performed without any filtering to depict commonalities and differences between the single categories and the entire source code. Based on this information we then compared every category with the unfiltered results to investigate significant influences of each category. To answer RQ1, how the different code categories are distributed within the test environment, we checked every category for its size counted in LOC. Following quality metrics were then selected for our investigation approach in RQ2:

1. unit coverage
2. comment ratio
3. method length
4. nested depth

Every metric was individually analyzed in every category as well as in the entire project. How the metrics correlate within the categories and whether the categorization is necessary after all to produce significant information about the quality aspects of a software, is done manually by comparing and relating the results from these metric analyses.

### 5.4 Procedure

To extend our collection of generator tags we tied in with the results of our generator analysis in chapter 3. After removing every Java class applying to the filter of the previously found generator tags, we checked all remaining projects for generated code. This was done manually by going through their created clone reports and checking for unknown generator tags. Every new tag was then added to the existing collection.

The Qualitas Corpus was subdivided in the following categories:

- i. "all", which represents the entire source code without any restrictions,
- ii. "production code manually-maintained",
- iii. "production code automatically generated",
- iv. "test code manually-maintained", and

- v. "test code automatically generated".

In order to perform our quality analysis on every code category independently we used different combinations of the filtering heuristics described in chapter 3. The disclosure of the particular quality metrics is done via different analysis processors provided by ConQAT. The configuration file which holds the respective assembling of these processors was provided by the CQSE GmbH. The descriptions of the following processors are derived from the processor documentation of the ConQAT feature ConQATdoc [3] i) LOC: the determination of the size counted in LOC is done by the "LOCAnalyzer" processor. ii) Clone coverage: the "JavaCloneChain" processor handles the elaboration of the cloning attribute "clone coverage". iii) Comment ratio: the "CommentRatioAnalyzer" processor is used to determine the comment ratio at character level which relates the amount of characters used in comments to the total amount of characters used in the entire Java class. v) Method Length: the "LongestMethodLengthAnalyzer" determines the length of the longest method for each class. vi) Nested Depth: to disclose the maximum nesting depth of a Java class, ConQAT provides the "NestingDepthAnalyzer" processor.

Once the analysis of each category was finished, we compared every quality metric between the different categories. The selection of projects used for the analyses was chosen according to the respective code category. We analyzed only these projects which in fact contain the investigated category to avoid distortions of our final results. How many projects of our collection are contained in each category is summarized in Table 5.2.

	<b>Production</b>	<b>Test</b>
Manually-maintained	110 (100%)	95 (86.36%)
Generated	44 (40%)	16 (14.55%)

Table 5.2: number (and percentage) of projects per code category

## 5.5 Result

In our previous analysis in chapter 3 we collected 51 tags created by different code generators using clone-detection. Among the 47 yet not tested projects of the software collection, 19 projects contained generated code. Still, after applying the filter of generated code based on the existing tag collection, only 3 more generator tags were found.

In this section we illustrate the different quality metrics and how they are influenced by the different code categories.

### LOC

The proportional share of each code category is shown in Figure 5.1. Stack chart (a) shows the distribution of source code of the entire test environment among the different code categories which sum up to a total of 23.399.379 LOC. The share of manually-maintained productive code covers 17.341.342 lines and is therefore with 74,11% the most significant part of the software collection. Only 6,32% of productive code is automatically generated which represents a total of 1.478.280 lines. 19,16% which counts 4.483.328 LOC is manually-maintained source code for testing or documentation purposes. Automatically

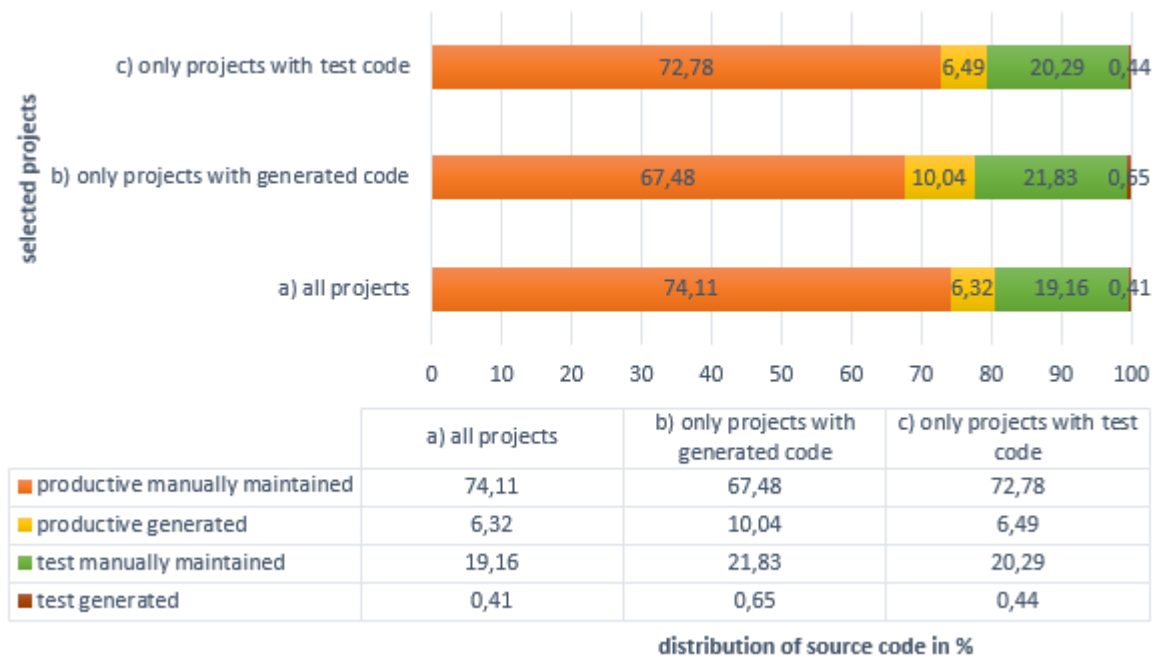


Figure 5.1: distribution of source code

generated test code only takes 0,41% (96.429 LOC) of the entire source code. As this small share of less than 1% doesn't affect the overall quality of a project, it can be considered irrelevant and will thus be excluded from our further analysis.

Stack chart (b) and (c) show the distribution of source code for only these projects of the software collection that in fact contain generated code, test code respectively. Both charts show very close resemblance to the code distribution within all projects, but with slight variations: stack chart (b) shows an increase of generated productive code to 10,04% and test code to 21,83% whereas the percentage of test code within only these projects containing test code increased to 20,29% (stack chart (c) ).

The reason for this is represented by boxplots depicted in Figure 5.2. Boxplots is a convenient way of graphically depicting the distribution of a dataset. The bottom, the middle and the top of the box represent the 25th (Q1), the 50th (median) and the 75th (Q3) quantiles. The whiskers represent the maximum and the minimum values that are not considered outliers. Outliers are represented by dots.

The displayed boxplots show the variations of the volume percentage per category of all analyzed projects. As one can see in boxplot 5.2(a), if analyzing the entire software collection, 50% of all projects contain at least 82,26% of manually-maintained productive code even reaching a top of 100%. The minimum as well as the lower quantile and the median of the percentage of generated productive code is 0, as over half of the projects don't contain generated code at all.

The median of test code varies from 11,5% (all projects 5.2(a)) over 13,82% (generated only 5.2(b)) to 16,43% (test only 5.2(c)). These small raises occur, as, by removing projects without generated code or test code, the highest proportion of deleted source code is manually-



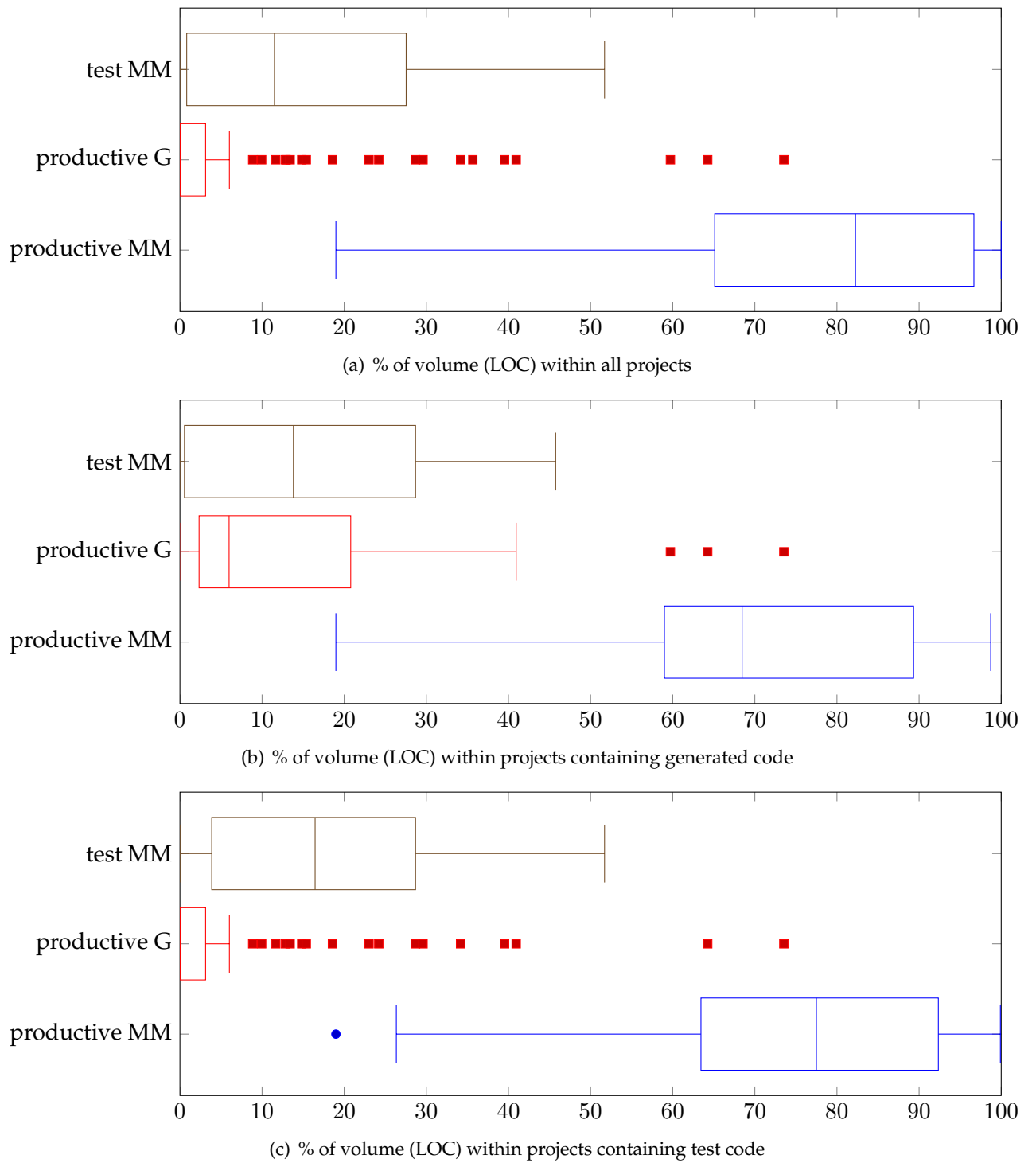


Figure 5.2: source code distribution among the code categories

maintained and for productive purposes.

The analysis of only projects with generated code reduces the amount of analyzed lines from 23.399.379 to 14.731.005 which is 62,95% of the original amount. This leads to a decrease of manually-maintained productive code from 17.341.342 lines to 9.940.158 lines which represents 57,32% of the original amount whereas the amount of manually-maintained test code was reduced only from 4.483.328 to 3.216.138 (71,73% of the original amount). The strong similarities between the evaluation of all projects and only these projects containing test code occurs as there were only 15 projects within the entire test environment that didn't contain any test code; therefore, the amount of analyzed lines was decreased to 22.101.576 which still represents 94,45% of the original value.

### Clone Coverage

The quantitative results of the clone detection analyses are depicted in Figure 5.3. The average clone coverage of all code categories is 26,25%. It ranges from 2,3% to 54,8% with a median value of 23,55%, which lies closely within the median value of manually-maintained production code (18,1%) and test code (25,25%).

The clone coverage of automatically generated production code has a very wide variation range from 0% to 98,1%. Its median value is 50,05% which is 32,4% higher than the median of manually-maintained production code.

The median value as well as the variation range (0% to 73,2%) of test code closely resembles to the clone coverage of the entire source code.

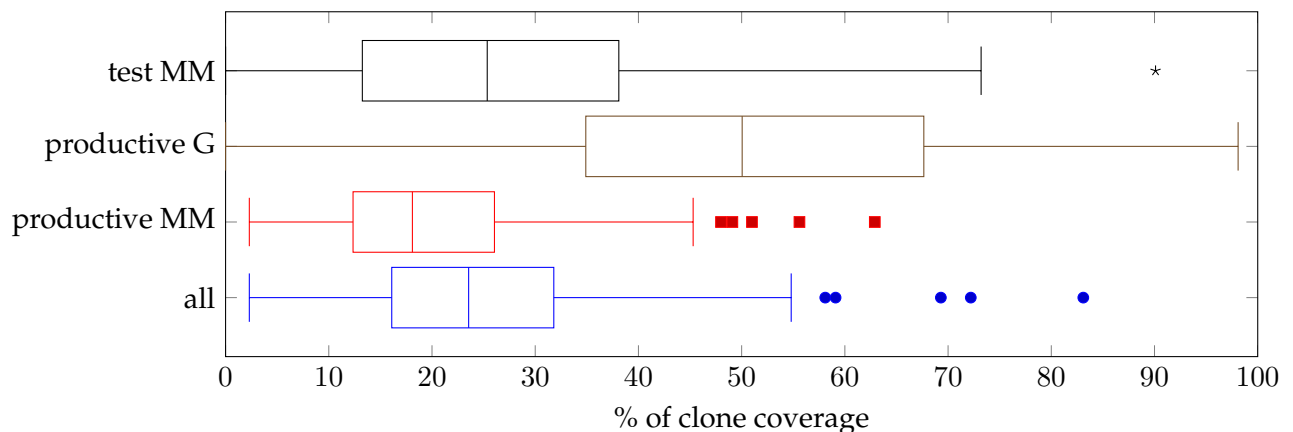


Figure 5.3: clone coverage of all code categories

### Comment Ratio

Figure 5.4 shows the evaluation of the comment ratio analysis. The ratio within the entire source code ranges from 22,2% to 70,04% with a median value of 46,84%. The analysis of manually-maintained production code with a median value of 50,42% and a range from 23,67% to 79,73% shares great similarity with the comment distribution of the entire source code, but with slightly higher values.

Within generated code the results show a decreased comment ratio with a median value

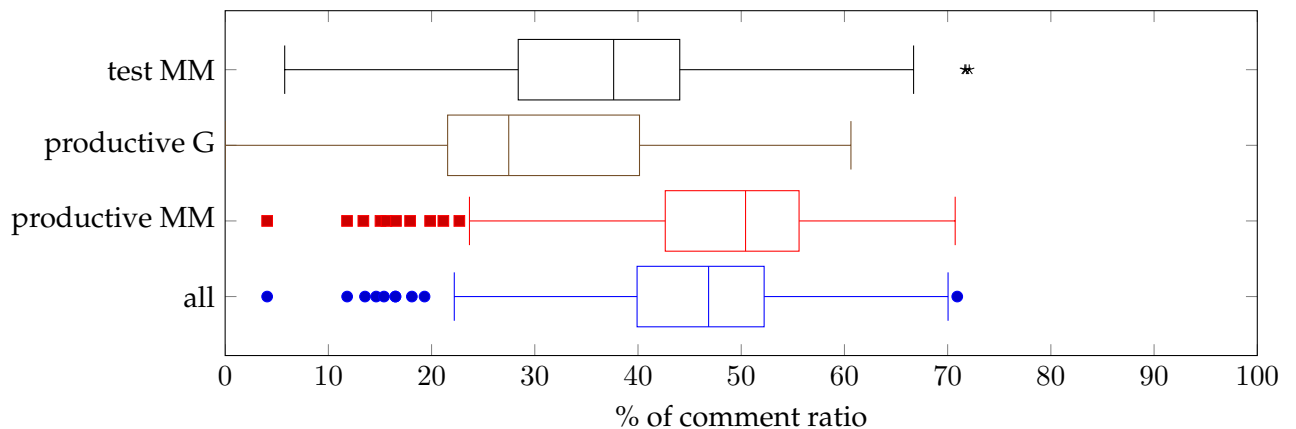


Figure 5.4: comment ratio of all code categories

of 27,47% and a range from 0% to 57,5%.

The comment ratio within test code varies from 5,77% to a maximum of 66,7% and a median value of 37,64%.

### Longest Method

The evaluation of the longest method analysis is illustrated in figure 5.6. The assessment of this quality metric is performed by ConQAT by classifying the outcoming results in disjunctive groups. As shown in figure 5.5, there exist 3 different quality grades a method can

**LSL**

Interval	Files	%Files	LoC	%LoC
]0;40]	5.851	82,0	851.505	55,3
]40;100]	967	13,6	384.818	25,0
]100;1.945]	316	4,4	303.686	19,7
Sum	7.134	100,0	1.540.009	100,0

Figure 5.5: classification of long methods

be assigned to: good, sufficient and bad quality. The assignment is done by counting each line that is not blank and not commented. Every method containing less or equal than 40 LOC is considered well-programmed and represented by the color green. More than 40, but less or equal than 100 LOC stands for sufficient code quality and is colored in yellow. Bad quality is considered every method containing more than 100 LOC, represented with the color red. Every diagram in figure 5.6 shows the proportional shares of each quality grade within the different code categories using boxplots.

#### i) good quality

The percentage of well-programmed methods in each category is depicted in diagram 5.6(a). Within the entire source code at least 12,8% and at most 93,2% of the existing methods contain less than 40 lines. Its median is 47,95%. Very similar with a median

of 48,15% and a slightly wider variation from 6,8% to 43,3% is the percentage in manually-maintained production code. The median value within automatically generated production code is with 25,6% 22,55% lower than manually-maintained production code, even though its variation ranges from 0% to 100%. This results from the fact that the proportional shares of half of the analyzed project lies between 4,5% (lower quantile) and 54,05% (upper quantile). The distribution of the percentage of well-programmed methods within test code behaves inversely. Despite the large variation from 7,3% to 100% its median value of 65,45% is 17,3% higher than the median in manually-maintained production code. This relies on the fact that the proportional share of well-programmed methods within test code lies between 46,5% and 80,85%.

### *ii) sufficient quality*

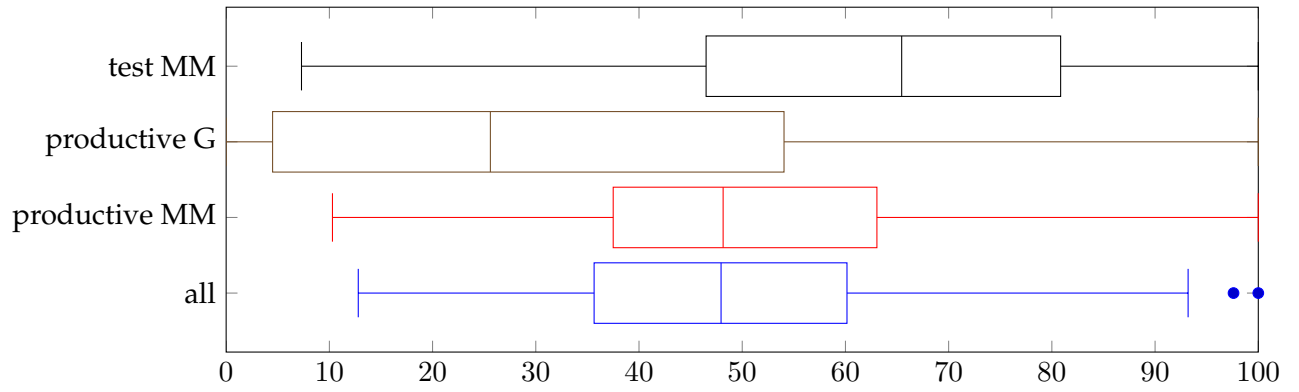
Diagram 5.6(b) shows the percentage of methods within each code category containing at least 40 and at most 100 LOC representing the sufficient quality grade. Within the entire source code the proportional share of methods with sufficient quality varies from a minimum of 6,5% to a maximum of 43,3% and a median value of 25,8%. Almost the exact same results shows the analysis of manually-maintained production code with a variation range from 6,8% to 43,3% and a median of 26,4%. Unlike in 5.6(a) the maximum of the variation within automatically generated production code decreases from 100% to 50,6% and the upper quantile from 54,05% to 28,6% which lowers its median to 12,05%. The percentage of sufficiently-well programmed methods within test code reaches from minimum of 0% to at most 62,5%. Its median value is 24,05%.

### *iii) bad quality*

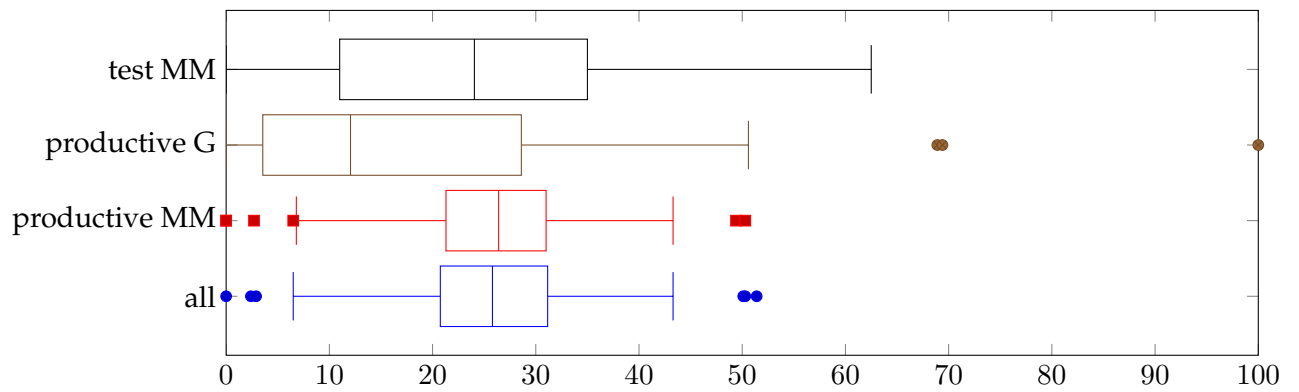
The proportional share of methods with more than 100 LOC is depicted in diagram 5.6(c). The analysis of the entire source code shows that at least 0% and at most 60,3% of methods within a project are considered badly-programmed, resulting in a median value of 21,7%. Almost the same results apply for manually-maintained production code. Its percentage of badly-programmed methods, with the same minimum of 0% and a maximum value of 60,1%, has a variation range that differs from the analysis of entire source code only by 0,2%. This leads to the slight decrease of the median value to 21,3%. The distribution of the proportional shares within generated production code spreads from 0% to 100%. With the lower quantile at 20,4%, the upper quantile at 81,45% and the median at 51,25% the results are homogeneously distributed. Within test code the median value is 6,3%. This results from the distribution of the quantiles; the percentage of three quarters of the analyzed projects is less than 16,55% (upper quantile). The variation range within this category reaches from a minimum of 0% to a maximum of 39,5%.

## **Nesting Depth**

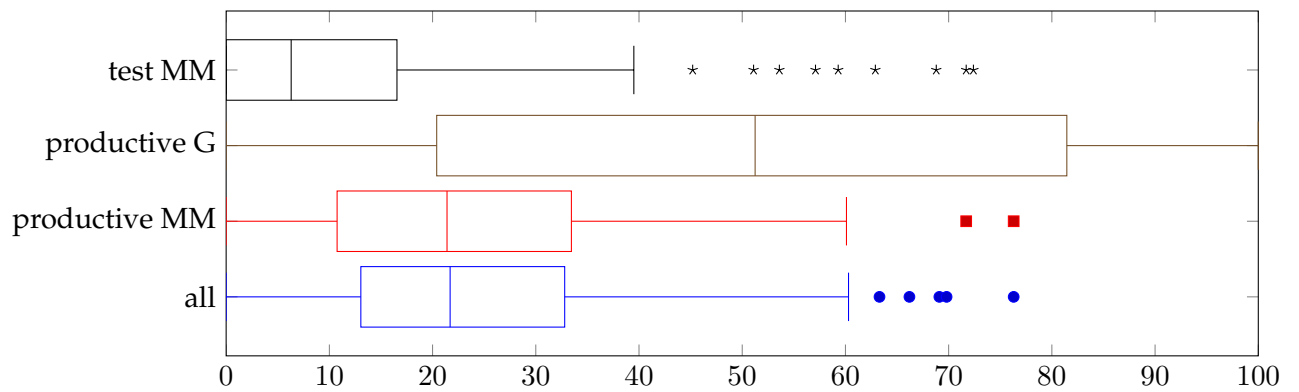
The evaluation of the nested depth analysis is depicted in Figure 5.8. The classification of this quality metrics in disjunctive grades is again performed by ConQAT. Figure 5.7 shows how the control flow analysis was assessed by ConQAT. Well-programmed is considered every conditional statement with less or equal than 3 decision points. Sufficient quality means every control flow with more than 3 and less or equal than 5 decision points. Every control flow exceeding 5 decision points is considered badly-programmed.



(a) methods with less or equal than 40 LOC



(b) methods with at least 40 and at most 100 LOC



(c) methods with more than 100 LOC

Figure 5.6: method length of all code categories

### Nesting Depth

Interval	Files	%Files	LoC	%LoC
]0;3]	6.469	90,7	1.155.477	75,0
]3;5]	531	7,4	269.436	17,5
]5;16]	134	1,9	115.096	7,5
Sum	7.134	100,0	1.540.009	100,0

Figure 5.7: classification of nesting depth

#### *i) good quality*

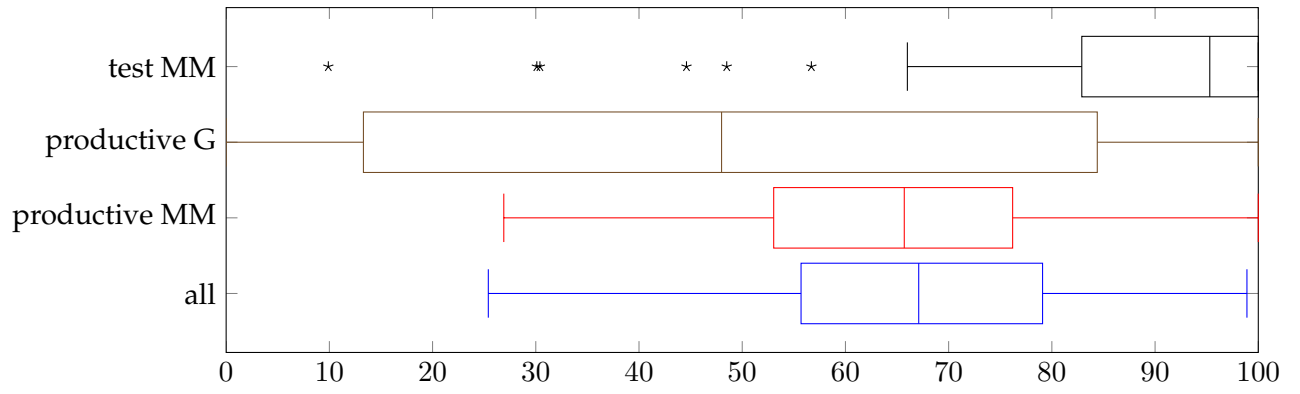
Diagram 5.8(a) represents the well-programmed percentage of conditional statements within each category. The entire source code has a median of 67,1%, contains at least 25,4% and at most 98,9% of conditional statements granted as well-programmed. Manually-maintained production code shares the same percentage distribution as the overall analysis with slight differences: the variation range starts from 26,9% and tops at 100%. Due to a decrease of the lower and upper quantiles the median value was lowered by 1,4% to 65,7%. Automatically generated production code shows a very widely spread result distribution from a minimum of 0% to a maximum of 100% as well as a very evenly spread distribution with the lower quantile at 13,3%, the upper quantile at 84,4% and the median value at 48,0%. Unlike the other categories there is a very high proportional share of well-programmed control flows within test code. Although there exist some outliers with the lowest value at 7,3%, the lower whisker has a value of 66,0%. The median is 95,3% which indicates that within half of the analyzed projects at least 95,3% of the conditional statements contain at most 3 decision points.

#### *ii) sufficient quality*

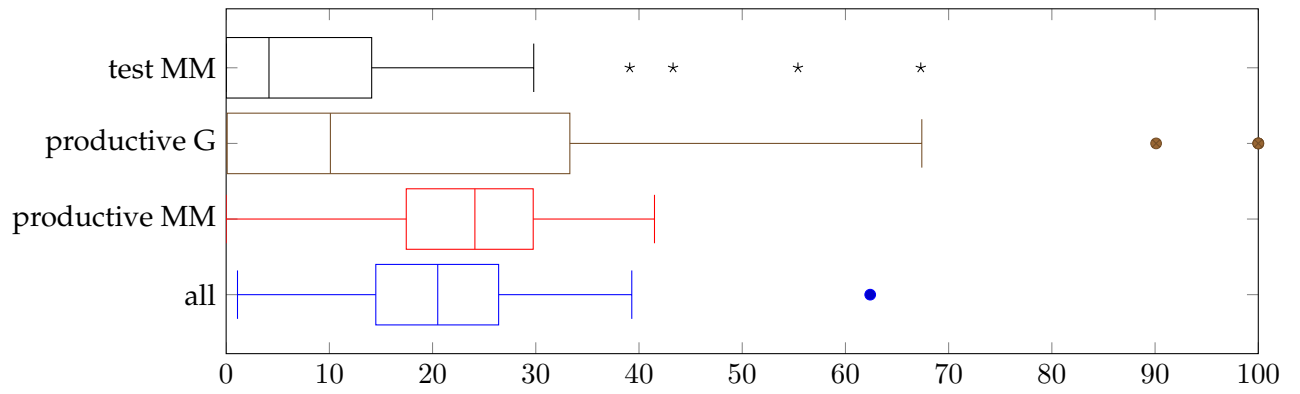
The percentage of control flows with more than 3 but less or equal than 5 numbers of decision are depicted in diagram 5.8(b). Within the entire source code the median summarizes to 20,5% as the results vary from 1,1% to 39,3%. Slightly wider spread are the results within manually-maintained production code: starting from 0% it reaches to a maximum of 41,5%. As the values of the lower and upper quantiles are with 17,45% (Q1) and 29,75% (Q3) 2,95% respectively 3,35% higher than the quantiles of the analysis of the entire source code, the median value also raises by 3,6% to 24,1%. Within automatically generated code, although there exist outliers at 90,1% and 100%, the upper whisker is set to 67,4%. Despite this proportionally high value, the median is 10,1% as the lower whisker is 0% and the lower quantile 0,1%. Even smaller is the result distribution within test code. Its median value is 4,5% as its upper whisker reduces the spreading to a maximum of 29,8% (outliers excluded).

#### *iii) bad quality*

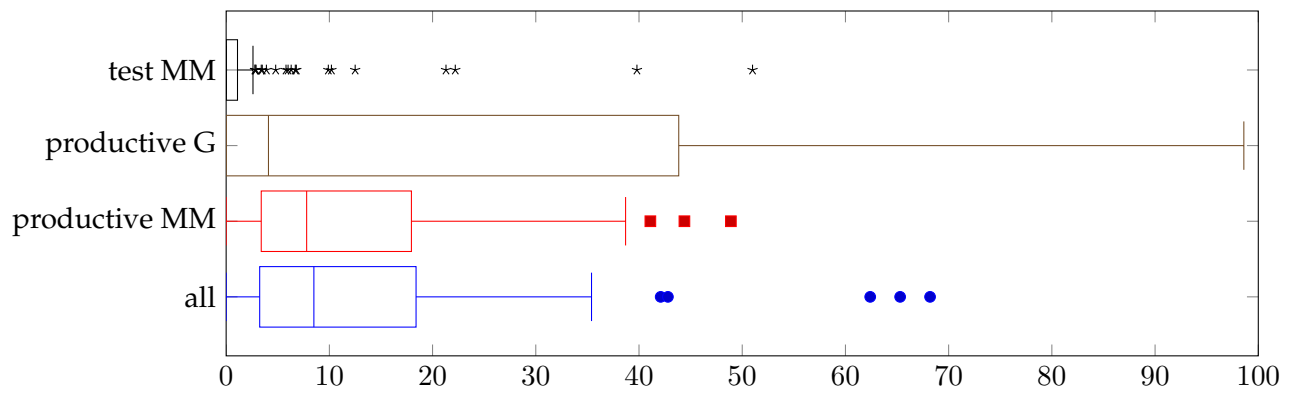
The proportional shares within each category of control flows with more than 5 decision points are presented in diagram 5.8(c). The analysis within the entire source code shows a median of 8,5%, a minimum of 0% and the upper whisker at 35,4%. Despite this low percentages there are several outliers which reach a maximum value of 68,2%. The me-



(a) control flow with less or equal than 3 decision points



(b) control flow with at least 3 and at most 5 decision points



(c) control flow with more than 5 decision points

Figure 5.8: nesting depth of al code categories

dian within manually-maintained production code is 0,7% smaller (7,8%) than the median value of the analysis within the entire source code, even though the upper whisker within manually-maintained production code is 3,3% higher (38,7%). The variation width in generated production code ranges from 0% to 98,6%. Even though the upper quantile is set at 43,85%, the median value is 4,1% which includes the lower quantile, whisker respectively to be 0%. The upper whisker of the analysis within test code limits the variation range of badly-programmed conditional statements to 2,6%. Median as well as the lower whisker and quantile are 0%. Still there exist a large variety of outliers which range right from the upper whisker to a top of 51%.

### 5.6 Discussion

We introduce this section by shortly evaluating the efficiency of the identification of generated code using the existing collection of generator tags, followed by the evaluation of the previously described results from our quality analyses. Every metric will be evaluated and discussed one after another followed by a final conclusion.

In our approach of identifying generator tags we determined that almost every generated code fragment was already detected by searching for the 51 saved generator tags. Although there were 19 out of 47 projects with generated code, only 3 generators weren't yet known to the applied filter. This result leads to the assumption that by continuously increasing a database of generator tags, the probability of exposing unknown tags decreases which appears to be a promising approach to automate the identification of generated code. This question about the efficiency of such a collection exceeds the research questions of this thesis though and will thus be mentioned in chapter 6 "Future Work".

#### LOC

How much influence a code category has on the overall quality of a software, relies on its proportional share. Summating the source code of all projects, manually-maintained productive code covers with 74,11% two-thirds and is thus the most significant code category. Even within the filtered project collections (see Fig 5.1.b, 5.1.c) its hardly decreased percentage makes manually-maintained production code the category with the most influence. As shown in Figure 5.2 though, there exist projects with minor significance of manually-maintained production code as its percentage goes down to 18,99%.

With approximately one-tenth share of the entire source code of all projects that contain generated code, automatically generated production code appears to have a minor influence on the overall quality of a software. Figure 5.2 illustrates the large amount of outliers of the proportional share of generated production code that reaches a top of 73,53% within a project. This percentage covers approximately two-third of the respective software and performing quality analyses without considering it leads to results that have a great risk of being erroneous.

Disregarded the projects selection used for the analyses, manually-maintained test code always covered approximately 20% of the entire source. Its proportional share within a project differs from 0% to 51,7% which generally makes it a reasonable part of software.

Automatically generated test code only covers a very small percentage of less than 1% of the source code within the different project selections. This and the fact that there weren't



any outliers with higher shares of generated test code makes this code category irrelevant for the overall quality of software and thus justifies the decision of excluding it in the analyses of this thesis.

### **Clone Coverage**

We determined an even distribution of clone coverage from 2% to 54,8% within a project excluding existing outliers. The resulting median of 23,55% is thus very similar to the average of all projects of 26,25%. These results are based on the influences of the different code categories. The clone coverage within manually-maintained production code resembles closely to the coverage within the entire source code, but generally tends to have smaller values. In contrast, automatically generated production code has an increased percentage of cloned statements. As the clone coverage distribution of test code is also very similar to the entire code, it is the impact of the comparatively small category "generated production code" which raises the clone coverage within all categories.

### **Comment Ratio**

Within all code categories we determined a very wide variation range of comment ratio. The results within the entire source code share great commonalities with the manually-maintained production code. They both show results with very high as well as with minor ratios which summarizes to a median of approximately 50%. Manually-maintained production code has a slightly increased comment ratio. The decrease of the ratio within the all source code can be traced back to automatically generated production code as well as test code. Despite the wide variation range of these two categories, their median values of 27,47% (generated production code) and 37,67% (test code) show a worse comment behavior within these code categories than within manually-maintained production code. Especially generated code contains a much reduced amount of comments reaching to a minimum of 0%.

### **Long Method**

The distribution of long methods within all source code has a large variation range. Although there exist projects whose methods are all considered to be well-programmed (contain less than 40 LOC), there are also projects with only 12,8% of good quality methods and 60,3% of bad ones. Manually-maintained production code shares the same distribution which summarizes to an average of 49,18% of good, 25,87% of sufficient and 24,95% of bad methods within this code category. Even better quality promises the category test code. Despite its wide resulting variation range from 7,3% to 100%, its average of 62,69% of good quality methods raises the overall quality of the evaluation of the entire source code. Although generated production code also shares a large variation range, its result distribution behaves inversely. With only 32,13% of good methods but 48,48% of bad ones, it lowers the result of the overall analysis of a project.

### **Nesting Depth**

The results of the nesting depth analysis have relatively high quality results within the different code categories. Especially within test code, on average 88,53% of the depth of conditional statements remains below 4 decision points. Generated production code shows results in all variations. Its average percentage of good control flows is 49,77% which is

equal to its median. Both other quality grades have comparatively low median values (10,1% sufficient; 4,1% bad) but very large result variations. Whether generated production code improves or worsens the results of this quality metric is difficult to investigate due to this very large but even result distribution. It is depending on each project itself, which includes projects with a very high percentage of shallow as well as of deep control flows. Nevertheless, as the average values of the different quality grades are 49,77% of good, 22,97% of sufficient and 27,26% of bad control flows, the quality of the conditional statements within generated code have a tendency towards the quality grade "well-programmed" rather than "badly-programmed". Compared to the average result distribution of 66,77% within the entire source code, manually-maintained production code shows a slightly decreased average of 64,79% of well-programmed control flows. It appears that this deviation is based especially on the different depth behavior of test code.

### Conclusion

In our analyses we showed the different quality behavior within each code category. Every quality metric showed great deviations between each category which emphasizes the need of categorizing source code in order to produce significant results about the quality of a software. Most of the times it was manually-maintained production code holding the largest share of a project. This leads to its strong similarity and thus impact on the overall quality within our analyses. Nevertheless we also provided evidence for the existence of projects with a very different percentage distribution of each category. These projects are obligated to be categorized first in order to consider only these values that are relevant for the quality evaluation of the respective software.

## 5.7 Threats to validity

This section contains potential threats to the validity of our analyses. Generally there were some issues within the categorization process that might lead the respective filtering approach to misbehave. We based our code identification heuristics on two main criteria: i) the location of a file and ii) certain identification marks within a file. Thus, every file containing one of the previously detected generator tags is automatically considered generated as long as those generator tags are still present. If there are generated files though that are modified and subsequently maintained manually after their creation, they are still recognized as automatically generated. Furthermore we don't distinguish between files that consist of several code categories. If a file contains code fragments that apply to one of the particular heuristics, the entire Java class is categorized respectively without considering the remaining class content.

As we base the categorization on heuristics there is no guarantee for the completeness of the resulting outcome. The patterns used to classify source code were manually tested and optimized during this thesis and can hence perform only with a certain degree of accuracy. The identification of code generators using clone-detection is an approach that has no evidence of validity. It presumes an increased clone coverage within generated code which does appear within our test environment but needs further research to be verified properly.

The benchmarks used for the optimization process of the filter heuristics are mostly created by examining manually potential Java classes which is no guarantee for their completeness. Also the consultation of the respective developers of the analyzed projects, although it brought large improvements to the accuracy of the benchmark, doesn't provide the assurance to cover every generated file; even though many developers cooperated with their help, it was not possible to get in touch with each development team.

Another threat to the validity of our study is the potential incompleteness of the analyzed projects. It is very common that especially generated files aren't uploaded to the repository and evolve only during the compilation process.

Furthermore we had to vary between the projects selected in order to optimize the analysis results. The evaluation of the manually maintained productive category was performed on a test environment consisting of 110 projects whereas generated productive code was tested only on 42 projects. Although the analyses showed a very similar distribution of the results, the occurring differences are a potential threat to the validity of this study.

## 6 Related Work

This section gives a brief overview of the current state of the art of code categorization. We describe a similar approach and extend our focus on cloning and the use of clone-detection.

### 6.1 Code Categorization

In his study "Categories of Source Code in Industrial Systems" [1] Tiago L. Alves introduces a categorization approach of source code, followed by an empirical study to illustrate its importance. His study contained a total of 80 systems provided by different companies with sizes ranging from near 5K to 1.7M LOC. Like in our study, the categorization of source code is structured with the two basic categories *productive* and *test* code but more specified within the sub-categories. Instead of reducing the differentiation to *manually-maintained* and *automatically generated*, T. Alves subdivides the two basic categories into 4 different groups respectively: *manually-maintained*, *generated*, *library* and *example*.

In this context Library code implements every code artifact that is developed and maintained by third-parties but necessary for the system logic or for testing. Example code implies every code fragment that does not have any productive or testing functions and its solely use is for demonstrating purposes, e.g. templates for developing new code. This differs from our classification of source code as we defined test code to be both, for testing and demonstrating purposes; hence the sub-group "*example*" is included in our main category "*test code*".

In T.Alves study, the identification of each code category was performed by applying certain heuristics. These heuristics are mostly based on a manual examination of the naming of the folder structure and the consultation of available documentation. Furthermore regular expressions were used to look through files for respective identification marks (e.g. "*generated by*"). These results were validated by consulting the product owners of the different systems. This approach of filtering the different categories resembles to the heuristics we used with certain differences: Most importantly, the categorization process within our analyses was performed automatically by ConQAT which requires the optimization of the precision of the heuristics. In order to increase this precision, we performed several test runs to optimize the selection of the respective strings that are decisive for each code category. Especially the identification of generated code is performed in a different way. Our analyses produced a large number of false positives by simplifying the search string to "*generated by*" which led to use of clone-detection to identify non-ambiguous generator tags.

Nevertheless, the results of T.Alves' empirical study show great similarity to the results we gained. Table 6.1 shows the number (and percentage) of projects per category of our analysis, table 6.2 the distribution of T. Alves' project collection. One can see the great similarity within manually-maintained productive and test code. The higher score rate of

	<b>Production</b>	<b>Test</b>
Manually-maintained	110 (100%)	95 (86.36%)
Generated	44 (40%)	16 (14.55%)
Library	-	-
Example	-	-

Table 6.1: number (and percentage) of projects per code category of our study

	<b>Production</b>	<b>Test</b>
Manually-maintained	80 (100.0%)	74 (92.5%)
Generated	52 (65.0%)	15 (18.8%)
Library	30 (37.5%)	7 (8.7%)
Example	2 (2.5%)	0 (0.0%)

Table 6.2: number (and percentage) of projects per code category of T.Alves study

generated code within T.Alves study could either be the result of the different identification approach or the use of a different collection of projects.

## 6.2 Cloning and clone detection

In this thesis, code cloning takes a major part in the identification process of generated code. Nils Göde et al. evaluate in their study "What Clone Coverage Can Tell" [4] the validity of the quality metric "clone coverage" of a project and describe three different parameters that have a large influence on it:

- i. the minimum length of a clone,
- ii. the exclusion of generated code and
- iii. whether identifiers and literals are normalized or not.

Figure 6.1 illustrates the deviations of clone coverage by just adjusting one of the three parameters. This experiment was performed on an industrial C/C++ system with 1.400.000 LOC (600.000 LOC when generated code is excluded). The results showed that clone coverage can vary between 19% and 92% depending on the chosen parameters. Leaving the other two parameters unmodified, whether or not generated code is excluded changed the outcoming coverage by approximately 10%.

Relating to our study, this emphasizes the importance of categorizing source code in order to produce significant outcome. By adjusting the parameter "normalizing identifier and literals", the coverage can change up to 50% depending on the distribution of the remaining parameters. The evaluation of these results led to the conclusion that clone coverage always has to be interpreted with care. Especially when comparing the coverage among different systems, it is mandatory to use the same parameters.

All these results agree with the results from our analyses. During the identification process of generator tags, we ran several test runs in order to specify the right value for the parameter "minimum length of a clone class" to filter generated code. Within our analysis, even

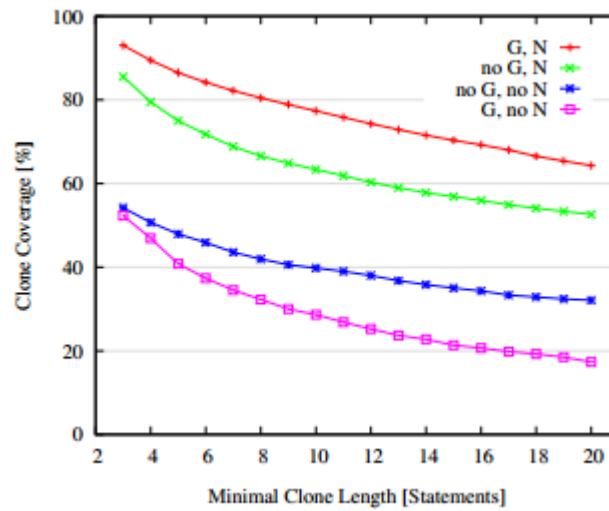


Figure 6.1: Clone coverage using different parameters (G = generated code is included, N = identifiers and literals are normalized)

though most code clones within generated code had a length of 10 or more, every code generator could be detected by setting 7 as minimum length. Consequently, most of the times generated code turned out to have the highest clone coverage within a project.

## 7 Future Work & Conclusion

In this thesis, we presented the importance of code categorization and how it can be accomplished. We hereby investigated several approaches of automating the categorization process. For this purpose, we introduced a new technique of identifying generated code using clone detection. By elaborating a compatible value for the predefined parameter "minimum length of a code clone", we were able to confirm the assumption that, in our software collection, the size of clone classes containing generated code is the largest or one of the largest of all clone classes of a system. By examining the Java files of the respective clones for generated code, we stored the identification tags of each code generator externally to be used later for the categorization procedure.

This database of generator tags was then tested for efficiency. As we enlarged this database, we determined that the ratio of unknown code generators rapidly decreases. This technique of detecting code generators was used among others to develop heuristics to recognize source code. Next to the clone-detection approach, they were based on commonly used names or, if existent, naming conventions or guidelines. The precision of the elaborated heuristics was verified by creating benchmarks of generated code and test code which were compared to the remaining source code after the filtering process.

After that we filtered every code category and independently performed several quality analyses in order to investigate the influences of each category on the overall quality of a system. The study revealed that at average 70% of a software is manually-maintained production code. Nevertheless, for some systems, the percentage of generated code and test code can represent up to over 70% and 50% respectively. Based on the investigated differences of the results of each quality metric among the code categories, we provided evidence about the risks of omitted or bad categorization. Mis-categorization can have a major impact on the measurements which involves a misleading view at the overall quality of the system. Based on these conclusions, there remain several interesting questions open which are considered to be treated in the future.

### *Creating a generator registry*

Up to now there exists no naming convention or guidelines to label generated code. Although it is very common to tag generated artifacts with the denotation "*generated by*", it is not sufficient to rely on this fact. In contrary, as illustrated in this thesis, one must not rely on such denotations as they hold a high risk of being used for other purposes. The most precise way of identifying generated code is the search for the exact phrasing of the labels of a generator. In order to spare the inconvenience of repeating the clone-detection approaches to identify the needed generator tags, we want to create a registry that contains a collection of the identification labels of each generator. As the amount of tags increases, the probability of detecting unknown tags decreases. This appears to be a great basis in order to automate the categorization process.

### *Automating code categorization*

In this thesis we developed a semi-automatically way of categorizing source code. This caused the analysis to be limited to a software collection of 110 projects which can still be examined manually. In his study "Amassing and indexing a large sample of version control systems: towards the census of public source code history" [10] A. Mockus describes the creation of a "universal repository for publicly accessible version control systems". To build this repository, he retrieved files from all publicly accessible forges which summated in a total of more than 200M distinct file-versions. Extending our test environment to a collection of systems with such size improves the validity of all research questions investigated in this thesis.

As it is impossible to check every file manually one after another in a collection of such size, executing our analysis on it requires automation. Therefore we want to put more effort into the research of automating code categorization. During our investigations we did not get further than a semi-automatically way, but with promising approaches. We need to detect unique features of each code category that makes them distinguishable from other code.

### *Verifying the identification of code generators using clone-detection*

Using clone-detection to detect code generators was a yet unknown approach. Hence, its validity still needs to be verified extensively. Although the results from the approach in this thesis applied to the existing benchmark of generated code, its settings (minimum clone length: 7; iterations to look for generated code: 4) are adjusted regarding to our test environment. One cannot be sure whether these values apply to every system. In order to verify this and to check for potential characteristics of generated code that make them recognizable among other categories, we want to extend our research in this field.

### *Extending the analysis to all programming languages*

Until now we limited our analyses to Java code only. In order to extend the evaluation of behavior and influence of the different code categories, we want to extend the analysis of this thesis to all programming languages. It may be interesting to see the variations of the relation between the categories and how they all interfere with the overall quality of a system.



# Bibliography

- [1] Tiago L. Alves. Categories of source code in industrial systems. Technical report, University of Minho, September 2011.
- [2] James R. Cordy Chanchal Kumar Roy. A survey on software clone detection research. Technical report, Queen's University at Kingston, September 2007.
- [3] Lars Heinemann Benjamin Hummel Elmar Jürgens Dr. Florian Deußenböck, Dr. Martin Feilkas. *ConQAT Book*, 2.7 edition, August 2010.
- [4] Nils Göde, Benjamin Hummel, and Elmar Juergens. What clone coverage can tell. In *Proceedings of the 6th International Workshop on Software Clones (IWSC'12)*, pages 90–91. IEEE Computer Society, 2012.
- [5] Volker Gruhn and Ralf Laue. Complexity metrics for business process models. In *in: W. Abramowicz, H.C. Mayr (Eds.), 9th International Conference on Business Information Systems (BIS 2006), Lecture Notes in Informatics*, pages 1–12.
- [6] Pekka Abrahamsson Hanna Hulkko. A multiple case study on the impact of pair programming on product quality. Technical report, 2005.
- [7] Elmar Juergens. *Why and How to Control Cloning in Software Artifacts*. PhD thesis, Technische Universität München, 2011.
- [8] Stephen H Kan. *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc., 2002.
- [9] Luigi Carro Luís da Cunha Lamb Flávio Rech Wagner Marcio F. S. Oliveira, Ricardo Miotto Redin. Software quality metrics and their impact on embedded software. Technical report, Informatics Institute, UFRGS, Brazil.
- [10] Audris Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 11–20. IEEE, 2009.
- [11] Martin Pöhlmann and Elmar Juergens. Revealing missing bug-fixes in code clones in large-scale code bases. In *Proceedings of the Seventh International Workshop on Software Quality and Maintainability (SQM'05)*, 2013.
- [12] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, December 2010.