UNIVERSITY OF GRONINGEN

RESEARCH PROJECT

Architectural smell detection using variable parameters based on project metrics

Author:
Jasper MÖHLMANN
Student number:
S1988794

First supervisor:
Prof. dr. ir. P. AVGERIOU
Second supervisor:
dr. V. ANDRIKOPOULOS
Daily supervisor:
Darius SAS

Faculty of Science and Engineering
Industrial Engineering and Management

May 18, 2019

university of groningen / faculty of science and engineering

Abstract

Architectural smell detection using variable parameters based on project metrics

This thesis presents research into the detection of God Components in software systems. Today, software is dealing more and more with Technical Debt introduced by e.g., Architectural Bad Smells. One of these smells is the God Component smell. Literature provides a wide range of definitions of this smell. In this thesis, a new definition is proposed, which includes the characteristic of relative size compared to other components in the system. Furthermore, a threshold-based detection method is proposed. The detection method is adaptive to the system it analyzes by selecting a system derived or benchmark derived threshold. The effectiveness of the proposed method is tested by implementing the adaptive detection method in Arcan, a smell detection tool for Java systems. The first results show good results by adapting to the system under analysis and detecting most smells present. False negatives however still arise, but the cause of this is the out-of-scope extraction of Lines of Code from reconstructed compiled Java files. Further work, therefore, could investigate a more rigid rebuilding of compiled Java files and include an analysis of the characteristics of the God Component itself to judge the actual impact of the detected God Component

Contents

Chapter 1

Introduction

In this chapter, an introduction is given concerning the thesis. In the first section, background information is presented on the topic of this thesis, namely architectural smells. Next, section 1.2 presents the problem description with a brief overview of the relevant topics for this thesis. The chapter ends with an outline of the rest of the thesis.

1.1 Background

Software patterns

In today's world, the number of software systems is increasing rapidly. Software is growing ever pervasive in every aspect of our day-to-day life. Businesses, government, entertainment, and even our social life are heavily dependant on software systems that grow in size and complexity every day. As these systems evolve, their architecture and internal quality degrades accordingly [4]. The size of these systems urge the need for a thought out architecture of software systems to keep them maintainable.
To support the development of large scale software systems design choices on an architectural level have to be made. On this level, system developers need to make design decisions on topics like communication, access control, synchronization, and data management. To support the developers, several architectural patterns in the field of computer science exist. Buschmann et al. define a pattern as

> *'A function-form relation that occurs in a context, where the function is described in problem domain terms as a group of unresolved trade-offs or forces, and the form is a structure described in solution domain terms that achieves a good and acceptable equilibrium among those forces.'*[5]

In the above-mentioned definition, Buschmann et al. describe already the strive for a balance between forces. Also addressed by Harrison et al., the decisions made for applying the pattern in the architecture can have an impact on the quality attributes [14]. These quality attributes, standardized in the ISO 25010 standard, support architects in selecting the appropriate patterns to fulfill requirements as set out by the customer of the system.

Technical debt

When developing a software system, it is sometimes necessary to sacrifice software quality in order to fulfill the needs of the clients timely. This trade-off between a gain in immediate customer satisfaction (i.e., deliver functionality) and sacrifice of the long-term quality of the system is termed as technical debt [7]. Over time, the accumulation of debt can eventually result in increased costs of maintenance and an increase in the technical difficulty of adding new functionality to the system or changing the current one. Avgeriou et al. later defined the term coined by Cunningham as:

> *'In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability.'* [2]

In the same proceeding, a conceptual model is depicted which list several causes for technical debt, including decisions, processes, artifacts, code, and defects.

Generally speaking, technical debt does present itself as, but not limited to, code smells and architectural smells. Code smells are mainly present at the implementation-level of the software. They impact classes, methods, and statements, and consist, for example, of duplicated code and long parameter lists. Although detection of technical debt in the form of code smells is rather straightforward, the reason for its existence could be on a higher level, at the software architecture-level. This level consists of components, connectors, and others. Technical debt on this level could be classified as architectural smells.

Architectural bad smells

Architectural smells commonly emerge when an architectural decision is not correctly applied, may it be due to the context of the system, the mixing of several designs or using the design on the wrong level of granularity [13]. The presence of an architectural smell will have a negative impact on the quality of the system and may reduce the maintainability of the software [17].
Although the negative impact of a smell on the system, its introduction can be intentional as well as unintentional. Where in the first case, the architect may be fully aware of the impact and still choose to implement it with its consequences, unintended occurrences are also common. The latter case is an emerging topic of research in the field of computing science.

In this thesis, the detection of a single architectural smell, the so-called God Component, will be central. The next section will elaborate on the problem context and definition.

1.2 Problem description

Automatic detection of architectural smells

When evaluating software, there are multiple methods to define its quality. Looking for code smells and architectural smells as well as metrics of a project can assist developers and managers in defining potential technical debt or problems. Whereas tools for the detection of code smells and the calculation of metrics are widely available, both as open-source and proprietary software, tools for the detection of architectural smells are scarce.
In a paper from Azadi et al. a catalogue is proposed that contains tools that can detect architectural bad smells. Their work focuses on three topics: *a)* what architectural smells are detected, *b)* what the definition of those smell is, *c)* how the detection is implemented in the tool [3].

In Table 1.1 an overview of tools analyzed by Azadi et al. is depicted.

Tool name	CD	HL	UD	CH	SF	GC	AwD	MH	AI	UA	ICD	AV
AI Reviewer	✓	✓		✓		✓			✓	✓		
ARCADE	✓	✓			✓	✓					✓	
Arcan	✓	✓	✓								✓	✓
Designite	✓	✓	✓	✓	✓	✓		✓	✓	✓		
Hotspot Detector	✓		✓	✓			✓				✓	
Massey Architecture Explorer	✓			✓			✓	✓				
Sonargraph	✓											✓
STAN	✓											
Structure	✓											✓

TABLE 1.1: Architectural smells detected by tools, adapted from Azadi et al.[3]

The smells addresses in the paper are: Cyclic Dependency (CD), Hub-like Dependency (HL), Unstable Dependency (UD), Cyclic Hierarchy (CH), Scattered Functionality (SF), God Component (GC), Abstraction without Decoupling (AwD), Multipath Hierarchy (MH), Ambiguous Interface (AI), Unutilized Abstraction (UA), Implicit Cross-module Dependency (ICD), and Architecture Violation (AV). As becomes clear from the figure, the support of different architectural smells varies between the tools. Only the Cyclic Dependency smell is detected by all tools.
The fact that all tools support it is not a coincidence, as it is considered one of the smells that affect maintainability the most in software systems [17]. The lesser support of others smells can be explained by the fact that the definition of an architectural smell is often subjective to the creator of the system and different tools can support the same architectural smell but with a different implementation [3]. This fact results in a weaker comparison between the detection possibilities of a tool and to the strengthening of the detection methods itself. No clear name and/or detection method is defined for some of the smells.

Arcan

As briefly mentioned in the previous sections, for this thesis, the Arcan tool will be central. Arcan is a tool being developed at the University of Milano-Bicocca by the ESSeRE group, lead by Francesca Arcelli, and work on the detection of architectural smells in Java software systems [11].
Since 2017, the development of Arcan continued, and support for more architectural smells was added, as shown in Table 1.1, with even more smells being under research to add in the near future, with one of those being the God Component smell.

The tool itself is built up around four core components within a layered architecture, which is depicted in Figure 1.1. The user can interact with Arcan using a GUI or a CLI to analyze Java *.class* or *.jar* files or a folder containing one of the aforementioned file extensions. The main processing unit from the figure contains all the logic for preparing the input and executing the smells detection. For storing information and providing the input data, a graph database is implemented, this allows to store the dependencies between the stored information easily. The Tinkerpop framework is used to 'easily build and access the dependency graph which represents the analyzed project and to allow the exploitation of different graph database backends.'[10].

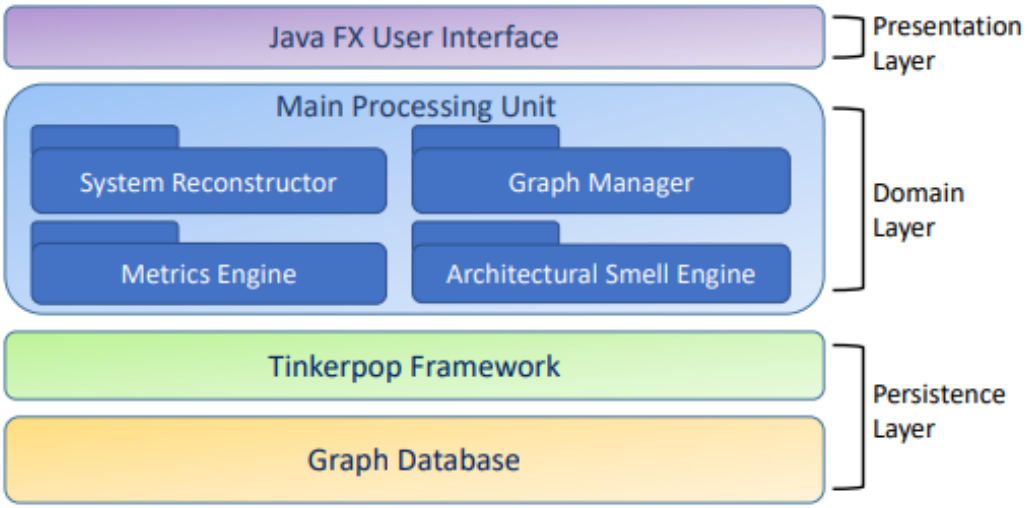

FIGURE 1.1: Architecture of Arcan, taken from [10]

The main workflow of Arcan is built around its four core components:

- **System Reconstructor**
  The System Reconstructor processes the submitted input in the form of *.class* or *.jar* files. Using the Apache Byte Code Engineering Library (BCEL) is extracts the original Java contents. The BCEL provides *JavaClass* objects which contain all the data structures, constant pool, fields, methods, and commands of a class object. The object itself is then made available to the Graph Manager for persistence storage.
- **Graph Manager**
  The Graph Manager translates the input from the *JavaClass* objects into graph nodes and creates edges that correspond with dependencies between the nodes,

such as a class-package dependency. Its other function is to initialize several properties of a node that later on can be filled by the Metrics Engine or the AS Engine.

- **Metrics Engine**
  The Metrics Engine computes metrics for the classes and packages in the graph database, which are later on used for AS detection. It computes metrics defined by Martin, such as Afferent and Efferent Couplings [20], and metrics defined by Chidamber and Kemerer, such as Fan In, Fan Out and LCOM, and stores these metrics to the nodes in the graph database [6].
- **AS Engine**
  The Architectural Smell Engine contains the logic of detecting AS by applying a detection algorithm to data contained in the graph database. Where required, thresholds are applied to prevent false positives, and the results of the detection are stored as separate nodes which are linked to the affected package or class node. Options are available to output the result of the detection engine to a folder when requested by the user.

God component detection in Arcan

One of the architectural smells for which Arcan is implementing in the AS Engine is the God Component smell. In the development version of Arcan, an algorithm is already in place that can detect God Components in the systems being analyzed. This implementation is based on a definition by Lippert et al., which defines God Components as excessively large components. To objectify this definition, they present a 'rule of 30', which describes a threshold based on the sub-components in a component. The scale on levels of granularity to determine what the sub-components of a component are. The levels of granularity they provide are System, Package, Class, Method, and Lines of Code (LOC). When a component contains over 30 subcomponents, for example, a Class with over 30 Methods, the class is considered a God Component. Multiplication can also be used if the metric for analysis is not directly present, for example, a class should not contain more than 900 Lines of Codes (30 methods multiplied by 30 LOC per method) [17].
Originally, this definition was never intended to be used in automatic smells detection but rather to propose an intuitive limit for the developer to consider as a benchmark when developing software. Arcan, and also Designite use this definition to detect the God Component smell[1].

Besides the drawback mentioned above, another shortcoming is that it reduces the reliability of this detection algorithm: when analyzing systems of different sizes, the algorithm does not take into consideration the average sizes of the components. Systems that may well be built to contain substantial components due to the nature of the system will always show up as God Components, when in fact this would be false positives as the developer intended to build significantly sized components. Context and characteristics of the system analyzed are not taken into consideration in the detection process.

[1]Does Your Architecture Smell? http://www.designite-tools.com/designite/does-your-architecture-smell

Problem definition

Considering the previously mentioned drawbacks of the implementation of the God Component detection as it is currently in Arcan in combination with the difficulties in properly defining a God Component as shown by Azadi et al. [3] the following problem definition is formulated:

> *There is no sufficient detection method in the literature for the God Component architectural smell which considers the context of the analyzed system.*

The problem definition underlines the difficulty of providing a solid definition of a God Component architectural smell. Besides that, currently, generalized algorithms discard the characteristics of the system that is analyzed. As Arcan is aiming at supporting more architectural smells, a new proposal for what Arcan classifies as a God Component in software should be presented in this thesis. Furthermore, the proposed definition and detection algorithm should be implemented into the Arcan tool itself for validation purposes and use to the users of Arcan.

1.3 Report structure

The rest of the thesis will deal with the problem description. In chapter 2 related work is presented. The literature on God Components and the derivation of the thresholds are presented. Chapter 3 outlines the methodology followed in the research. It presents the research goal, and research question and the approach followed to reach the research goal. The development of an adaptive detection method for God Components is described in chapter 4. Chapter 5 follows with a validation of the method developed in chapter 4. The thesis concludes with chapter 6, containing the conclusion and a future work section.

Chapter 2

Related work

In this chapter, a literature review is presented on related work with respect to the topics addressed in the previous chapter. In the first section, the set-up of the literature collection is given. The next section presents the results of the literature study.

2.1 Literature review set-up

To assist in the repeatability in the collection of the related work, the starting point for the literature study is documented below. A list of keywords and the search engines are outlined below. Some results were collected as web pages on the internet, as the documentation of many smell detection tools is presented on the websites for each separate tool.

The following keywords, or a combination of them, were used for an initial search:

- God Component smell
- Architectural Smell detection
- Architectural bad smell
- Threshold derivation
- Architectural smell threshold

These initial search terms were used on the following databases or search engines:

- Google Scholar
- IEEE Xplore
- Smartcat (University of Groningen internal literature search tool)
- Elsevier ScienceDirect

If any articles or books came up in the results, they were quickly scanned and, upon finding promising information, further examined. Relevant references in the literature results were also looked up for further investigation. The results of this search are summarized in the section below.

2.2 Literature results

2.2.1 God Component

As stated in the previous chapter, the God Component architectural smell is the main subject of study in this thesis. When examining the literature on this topic, a wide range of definitions and possible detection strategies are presented by different authors.

Azadi et al. present a catalogue of architectural smells in order to come to a single catalogue of smells within the academic world and industry. The God Component is one of the smells that is in their first proposal. They describe the smell as the "indication that a component implements an excessive number of concerns and accumulates too much control." In their classification of smells, the God Component is classified as a smell that violates the principle of modularity in a system. A summary of detection methods by several tools is included. The tools mentioned all use other detection algorithms, as some focus on detection on class level whereas others cover all levels of granularity in the software [3].

A widely cited work is that of Lippert et al. in their book 'Refactoring in Large Software Projects'. In contradiction with many other papers, they do not describe the characteristics of a God Component but rather define an element in which it is highly probable to be a serious problem. As already briefly mentioned Lippert et al. define a rule that is applicable on all levels of components in a system. More specifically, they provide the following thresholds to define an element which can cause issues [17]:

a) Methods should not contain more than an average of 30 code lines, not counting line spaces and comments.

b) A class should contain an average of less than 30 methods, resulting in up to 900 lines of code.

c) A package shouldn't contain more than 30 classes, thus comprising up to 27,000 code lines.

d) Subsystems with more than 30 packages should be avoided. Such a subsystem would count up to 900 classes with up to 810,000 lines of code.

e) A system with 30 subsystems would thus possess 27,000 classes and 24.3 million code lines.

Lippert et al. acknowledge that the issue of large elements that could cause problems can arise on multiple levels in a system. Their guidelines provide a method that can operate on different levels in a system using different metrics, as reported in the aforementioned list. The base metrics is the number of non-blank, non-commented lines of codes in the element under investigation. The base metric provides for a simplified implementation as the threshold for an element can be derived from its kind, be it a class, method, or subsystem. The tool Designite implements this strategy for detection on class level for lines of code and, for package level on the number of classes.

Another definition that fulfills the description of a God Component given by Azadi et al. is Lanza and Marinescu's definition. In their book 'Object-Oriented Metrics in Practice', a wide range of smells is described. Three of the smells could be classified as an example of a God Component. The smells are Brain Method, Brain Class, and God Class. The God Class smell is a design flaw that refers to a class that tends to centralize the intelligence in a system to its own. Almost all logic is executed in the class and other classes complete only trivial tasks. The God class can be detected by three characteristics [15]:

1. They heavily access data of other simpler classes, either directly or using accessor methods.
2. They are large and complex.
3. They have a lot of non-communicative behavior.

The Brain Method and the Brain Class differ from this description on the first and third characteristics, respectively: the God Class is almost solely responsible for all business logic, and data exchange, the Brain Method or Class only contain the business logic and no methods for data exchange. In Figure 2.1, the detection strategy for the Brain method is depicted. Lanza and Marinescu combine several metrics that determine if a method is a Brain Method. For each metric, different threshold levels are established so that smells could apply an appropriate threshold level for its detection strategy.

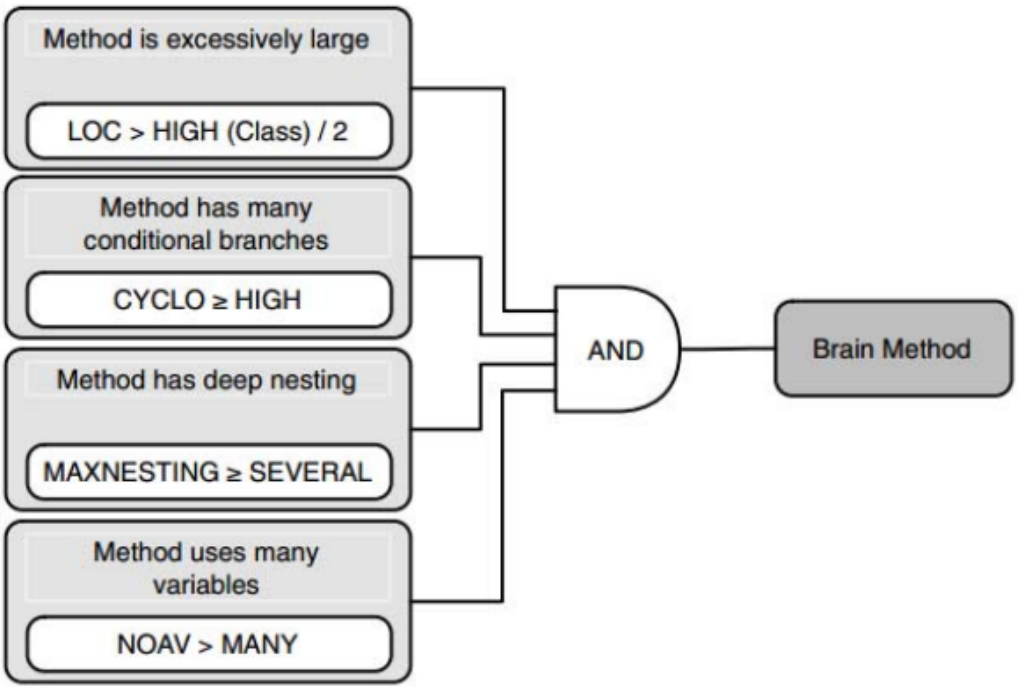

FIGURE 2.1: Brain Method detection strategy, taken from Lanza and Marinescu[15]

In short, Lippert et al. focus on the size of an element whereas Lanza and Marinescu focus on more metrics for detection.
Another approach studied by the literature is the number of concerns associated with an element. Le et al. propose a smell called Concern Overload, which is an

element which implements an excessive amount of concern. This characteristic results in a component that is large in size and may become difficult to maintain. The detection strategy is based upon the Number of Concerns per Component metrics, which is held against a user-defined threshold [16]. The detection strategy relies on the proper classification of concerns in the software analyzed. Some tools can help in executing this process, but often, the assignment is done manually, making the detection strategy a labor-intensive one [9].

2.2.2 Threshold derivation

Strategies for smell detection are rarely based on a binary judgment but often are based on a threshold for one or more metrics connected to the smell. This downside is, besides the definition and its related detection strategy, one of the main challenges in detecting architectural smells. Research that uses the proposed detection strategies with thresholds from the original authors often experience the problem of unreliable results. Thresholds are adjusted to fit the new systems being analyzed in the new research. In this way, the problem is not the detection strategy itself but rather the relationship between the threshold and the analyzed subjects.

When examining methods for threshold derivation, a wide range of approaches can be found. Threshold derivation can be done by using statistical analysis, professional experience, a correlation between variables like bugs and smells, metrics analysis, and methods that include machine learning and artificial intelligence.
The derivation by experience is one of the simplest methods to define a threshold for an algorithm but can be open for discussion on the robustness of the thresholds. An early example of the usage of an expert determined threshold is that of McCabe. He proposed a method for measuring software by its complexity instead of size. As a threshold for the distinction between a 'good' system and 'bad' system the threshold was set to a cyclic complexity of 10, but immediately acknowledge the fact that this limit can be inapplicable for some instances [21].

Another approach is deriving the threshold by using the metrics of the system. Using statistical metrics such as mean (μ), standard deviation (σ) and percentiles a threshold can be derived. Erni et al. propose a threshold derivation that produces a lower limit of $\mu - \sigma$ and an upper limit of $\mu + \sigma$ [8]. The formula is applied to the metric that the user wants as a threshold. An example would be the number of lines of code a class may contain. The average and standard deviation for all classes could be calculated and lead to an upper and lower limit to detect outlier classes. The downfall of this detection strategy is that the returned number of outliers is always 32% of the total population. The analyzed system will, therefore, always contain exceptions despite the size of the standard deviation and the 'correctness' of the examined elements. Furthermore, the strategy requires the data to be normally distributed in order to produce a correct output of the method. When applied on highly skewed distribution, the applied method would result in incorrect output that neglects the actual problematic elements.

A possible method for predicting issues, i.e., architectural smells, is to examine the correlation between metrics and the error itself. Shatnawi et al. propose a method that tries to define threshold by examining the correlation between several metrics and the presence of bugs a failure in three versions of Eclipse. Using the Receiver Operating Characteristic (ROC), they derived a threshold for several metrics which could be used for error prediction. The limitation of this approach is that the results are only applicable to the system which has been examined during the formation of the thresholds. In the study of Shatnawi et al. this effect shows up in the different threshold levels for a single metrics when applied on a different version of Eclipse [22]. This characteristic makes the applicability of the threshold in general low.

Today, more research is done into the use of benchmark-based derivation of thresholds. Using a dataset consisting of representative systems that can function as a benchmark in determining the threshold for a certain metric. Alves et al. propose in their work a method that works according to three fundamentals [1]:

i) it respects the statistical properties of the metric, such as metric scale and distribution;

ii) it is based on data analysis from a representative set of systems (benchmark);

iii) it is repeatable, transparent and straightforward to carry out

These fundamentals resulted in a derivation process that uses the weight and relative size of the systems in the benchmark set to represent the benchmark data uniformly for threshold derivation. After these steps, a predefined percentile, in the case of Alves et al. 90%, is selected for which the value will be the threshold. Different percentiles could be chosen to work with several levels of threshold, i.e., Low, High, and Medium, on the 70th, 80th, and 90th percentile. The limitation here is still the manual selection of a percentile set as the starting point for selecting the final threshold.

Arcelli et al. expand the method from Alves et al. by introducing a variable percentile for the threshold selection. Based upon the frequency distribution of 100 percentiles, they select the median value and search for the percentile for which all values are lower or equal to the median. By applying this method, the most repetitive values are filtered out of the data set, and only the more variable values remain. The selected percentile is used as a cut-off point for the original data set from the benchmark systems. On the cut-offed data set, again percentiles are selected to match Low, Medium, and High threshold levels, corresponding with percentile 25, 50, and 75.

Chapter 3

Methodology

In this chapter, the methodology applied in this thesis is described. The first section presents the research goal and the research questions that contain this research goal. The section that follows describes the steps which will be executed in this research in order to answer the research questions and reach the goal. The last section contains an overview of applied tools and techniques in the step from the middle section.

3.1 Research goal and research questions

To provide a solution to the problem statement as defined in chapter 1, a research goal is formulated. From this research goal, several research questions are defined which help in reaching the goal.
The research goal will be two-fold. First, it should deal with the development of a solid detection algorithm for the God Component. A non-static, but context-aware detection method should be developed. The second part is the implementation of the detection method into a detection tool for automated detection. For this thesis, the Arcan tools will be used. Based on this, the following research goal is formulated:

Develop and validate a detection method for the God Component smell that takes into consideration the size of the project itself and implements this method in the Arcan tool.

As already mentioned earlier, the current implementation in Arcan lacks an algorithm that considers the size of the project itself. Lippert et al.'s definition only provide a guideline on how big components should be rather than defining a God Component.

Research questions

To complete the research goal, several research questions are formulated. The research questions each focus on a specific part of the goal and help in providing a successful completion of the research goal.

The following research questions follow from the abovementioned research goal:

- **How to design an adaptive detection algorithm for the God Component**
  One of the limitations of the current detection algorithm implemented in Arcan is that it uses a fixed threshold for detection of God Components. The goal of this research question is to provide a method that supports the discovery of God Components in an adaptive way. By considering the system that is being analyzed, the detection algorithm should adapt to its characteristics and adjust the detection accordingly.
- **What changes are required in Arcan to implement the detection algorithm?**
  The current detection algorithm in Arcan has to be replaced by the developed algorithm from the first research question. This adjustment will require changes in Arcan to deal with the context-aware strategy that will be deployed. This question aims to support the implementation of the detection algorithm in Arcan by describing the required changes in code and architecture.
- **How accurate is the detection algorithm on widely used software systems?**
  The final research question focus on the accuracy of the implementation in Arcan. To validate the usefulness of the formulated detection algorithm and the implementation of the algorithm into the Arcan tool validation test should be performed.
  The implementation in Arcan should be validated and tested on the widely used software systems to see how the algorithm behaves when used in a practical and realistic environment.

The result of each question will be used as input for the next issue. After the last research question, a proposal for fulfilling the research goal will be ready and would be able to cope with the problem statement as defined in chapter 1.

3.2 Methodology

To perform thorough research, and to support its reproducibility, this section will outline the methodology adopted. The steps executed in the research are described below with argumentation for the step itself. Followed methods and selection of tools or techniques are listed for each step.

The research consists of the following parts:

- **Propose definition God component**
  The first step executed in this research will be the formulation of a definition for the God Component. This proposed definition will be used throughout the thesis as the guideline on how to detect God Components.
  In chapter 2, several definitions of a God Component have been presented. These definitions are suitable to use as a basis for the formulation of a God Component in this thesis. The added value of the definition proposed here is the link with the context of the system in which the God Component should be detected.
  The envisioned implementation in Arcan will be based on package level detection, but a broader applicable definition of God Components should be sought. This broader scope means that the definition should hold on all levels of granularity within a software system, being it on method level or system level itself. One or more metrics will be included in the definition to support objective detection in systems.

- **Create method for detection**
  The second step consists of the creation of a method for the detection of the God Component using the definition from the previous step.
  The method will describe what inputs are required for the detection, how these inputs are processed, and what the output should be. The step is about the translating of the definition into a method that is capable of detecting the God Component.
  The detection of a God Component will depend on thresholds that mark a component as a smell. Since the research goal aimed to provide an adaptive algorithm for this detection, the method will include ways to adjust the used threshold to adapt to the system it is analyzing. Argumentation for the implemented strategies will be provided.

- **Develop Algorithm for threshold derivation**
  As stated before, the classification of a God Components relies upon a threshold for one or more metrics. This step will provide an algorithm for the derivation of such a threshold based on characteristics of the system.
  To create an adaptive method, the threshold should be derived from the system itself rather than being a generally fixed threshold. In the literature, several possible derivation algorithms are proposed, which will act as background information in the development of this derivation process. The algorithm will consider the definition presented in the first step when deriving the threshold. It will be a generally applicable algorithm for deriving thresholds for the God Component smell.

- **Benchmark the model for a generally applicable threshold**
  The algorithm from the previous step will define a system threshold which will be between the minimum and maximum of the used metric of metrics in the detection algorithm. This characteristic results in a detection algorithm where God Components are always present.
  A benchmark study onto a set of representative systems should deliver a generally applicable threshold which can be combined with the system derived threshold to provide a robust detection method. Using the threshold derivation algorithm from the previous step, a benchmark will be extruded. Based on a set of Java systems, the algorithm will analyze the characteristics and extract a globally applicable benchmark.

- **Arcan implementation changes**
  The detection algorithm will require some changes in Arcan to detect God Components successfully. Changes in the way Arcan processes input files are foreseen as are adjustments to the graph database in order to provide data on all the required metrics required in the detection algorithm. A selection for certain libraries will be made that can help in the implementation of the algorithm. For this, the preference lies with open-source packages to fully understand the process from input files up to the classification of components as God Components.

- **Validate using manual search & Arcan detection runs**
  The final step is focused on the validation of the detection algorithm on Java systems.
  A selection of 5 open-source systems will be analyzed by Arcan to run the detection algorithm on the systems. The systems are also manually analyzed for God Components. The manual analysis will consist of comparing the results from Arcan with the system itself and also an examination of the biggest components in the source code independent from the results of Arcan.
  A comparison between the results from the manual detection and the Arcan runs will provide an overview of the correct defections, the false-positives, and false-negatives.
  The runs performed with Arcan also function as a test for the robustness of the implementation and the changes in the code.

The definition of the God Component, the detection algorithm, the threshold derivation, and benchmarking are outlined in chapter 4. The validation of the detection algorithm is described in more detail in chapter 5.

3.3 Tools, and techniques

During the execution of the steps mentioned in the previous section, several tools and techniques are used to aid the process. Listed below are the most important ones:

- **Java 8**
  Arcan is written entirely in Java version 8. The implementation of the detection algorithm should be able to work with methods and features available in version 8. Using features available in newer version would require restructuring more components and could also lead to conflicts with used dependencies.

- **Java source reconstructer**
  Arcan uses compiled Java .class and .jar files as input. The algorithm may require to use certain metrics that are available only when examining source code. Such metrics include the number of Lines of Code in the source code. The current decompiler in Arcan can extract data on the compiled classes but is not able to reconstruct editable source code in the .java format. For counting the number of lines of code, the program requires this function.

- **Python 3.6**
  During the development of the algorithm and the derivation of the benchmark threshold, larger data sets will be examined. Python provides easy-to-understand programming language with good support for data processing and analysis.
  To ease the process of data processing libraries such as Numpy and Pandas are used. These libraries provide features to process the expected data used in the algorithm development. Functions, such as data frames and several transformational methods, help in easy processing.

- **Benchmarking**
  Benchmarking is applied in the derivation of a generally applicable threshold for the God Component detection. Benchmarking will support the generation of a threshold that holds up for a wide range of systems types. Comparing the results of several systems from the benchmark aid in the applicability of the algorithm as a whole.

Chapter 4

Model development

In this chapter, the model development is described. The first section contains the formulation of a definition for a God Component, which is used in section two for the development of an adaptive detection method. The section that follows provides a basic set of steps in extracting the threshold used in the detection. The chapter concludes with a section on the derivation of a benchmark threshold.

4.1 God Component definition

When examining the different definitions of God Components in chapter 2, a wide range of definitions presented itself. Where some authors define a God Components as an element which is responsible for too many concerns inside the system, others focus on the metrics like the number of sub-components or access to foreign components.
Although all the definitions present substantial differences, and perhaps disagree in some cases, all of them agree on the fact that God Components are critical manifestations of technical debt.

Technical debt will be the starting point of the definition proposed in this thesis. Three topics are considered within the formulation:

- **Impact**
  The first topic is about the impact of a God Component on the system and on different quality attributes of the system.
  In literature, the impact of a God Component is described in multiple ways. Some authors focus on the negative effect a God Component has on the modularity of a system as the smell contains too many responsibilities and concerns. This characteristic causes an issue when trying to add new functionality to the system in later development iterations as features in the God Component are too intertwined to be used elsewhere or to be split into multiple functional segments [18].
  Other authors mention the concern of modifiability in their research. The God Component is in these cases often described with a size element in the definition. Large components reduce the effectiveness with which the component can be modified to fulfill current and future requirements [24].
  Many of the quality attributes affected by a God Component can be measured by the size of components. Modularity, modifiability and other maintenance aspects often degrade if the size of the component grows in size. Long names

can lead to confusion and reduce readability; long methods can be a sign of inefficient coding; long classes could be an indication of too many concerns clustered in the class. Following the steps of other authors to identify a God Component, this work will adopt the size of the component as well.

- **Granularity**
  In the literature, God Components are defined on different levels of granularity. An example of another name for the God Component, which has multiple appearances in the literature is the God Class. As the name gives away, this God Component is focused on the level of software classes. However, research on the level of package components is, by the knowledge contained in this research, not present. As a God Component can emerge on all levels, this research tries to develop an algorithm to detect God Components on this level of granularity.

- **Adaptiveness**
  Most definitions in the literature rely on fixed thresholds to define a God Component. If the size of a component i exceeds threshold x or component j containing more than y methods, the component is considered a God Component. These definitions result in a static definition, which is not able to adjust for the evolution of software projects over time nor to the context of the system itself. Software projects are growing rapidly in size, and a definition with a fixed threshold will result in more components being classified as a God Component over time due to this growth in project size.
  The definition of what is a large project strongly relies on the perception of the developers of that system. Developers working on very large scale systems may be experienced in dealing with large components for which no other options are possible and tend to consider their large-scale project as normal. Certain industries may also have this characteristic. In aerospace engineering, complex software systems with millions of lines of code and components may be the standard. The perception of the developer influences the definition of a God Component. Based on the number of lines of code or components, a developer may or may not consider the component to be a structural issue.
  The developer's bias and the project standard for size create the need for the introduction of an adaptive algorithm which can relate the context of the system to the classification of components as God Components.

Considering the above-mentioned, the following definition of a God Component is proposed:

A component whose size exceeds by far the average size (i.e., is an outlier) of all components in the system or the average size from the industry benchmark.

This definition provides a starting point for defining detection methods that should deal with detection on different levels of granularity, the impact of the size of a component, and context of the system it is analyzing.

4.2 Adaptive detection method

The definition of a God Component in the previous section provides a descriptive formulation. For the implementation of this description into an objective, repeatable method, several challenges have to be solved. Based on the definition, three components of the detection method will be further addressed:

- **How to measure size**
- **How to compare between different components**
- **How to classify a component as a God Component**

Size
In the definition, the main characteristic of a God Component is its size. When talking about size in software, many different metrics are proposed to define the size of a system or components of the systems.

The size of a component can be measured by looking at the number of components from a lower level it contains, for example, the number of classes in a package. This method provides a straightforward registration of size as it only requires a list with the different levels of granularity within a system after which for all components, the number of sub-components can be calculated. However, when reaching lower levels inside the system, such as classes, different sub-components become available. Modifiers, methods, variable declaration, and others create the problem of uneven impact. A variable does not hold the same weight on a component as a method that performs all communication with other classes or systems.
To tackle this problem, a single standard metric would solve the weight issue and could provide a variable to describe the size of a component. As the major drawback of God Components is an increase in the complexity of the system, we opted to use the number of lines of code as the base metric to measure the size (rather than the number of files for example). The number of lines of code has been demonstrated to be correlated with the complexity of a system [19]. Lines of codes form declarations, methods, classes and eventually, the system as a whole.
For measuring the size of a component, we will use the number of lines of codes in a component. The easy collection of these metrics, as well as the switching between different levels of granularity, make this metric a reliable and objective metric.

Formally, considered as lines of codes are any lines that contain text. Blank lines within the component are not considered an issue and are therefore not included in the metric. Commented lines do affect the readability, and excessive use of comments may lead to unclear code and therefore is included in the metric.

Comparison
As the definition relies on the relative size of a God Component, the need arises to register the same metric for all components. The detection method requires that for the chosen level of granularity, the input will consist of all the components on that level. For each component, the number of lines of code should be registered.
Clear documentation should be available on how the calculation of the number of lines of code is performed. Details on the included and excluded elements, the conversion from one level to another, and the possible translation of an input file into a

file type that support the reading of lines of code are hereby described.

In this thesis, the detection of God Component will take place on a package level. The input is a set of compiled Java classes which will run through a reconstructor. This reconstructor provides for each class the number of lines of codes and the package name. On class level, the lines of code count are stored. Following, for each unique package, the sum of the lines of code of all classes is calculated. The result of this calculation will function as input for the comparison itself.

Classification
The most critical part of the detection will be the classification of components, either as positive or negative when tested for the characteristic of being a God Component. In the definition, the term excessively is used in combination with the comparison between components of the same system and the comparison with a benchmark.
To turn the subjective term "excessively" into a scientifically meaningful threshold, the detection method must rely on empirical data and an empirically-validated methodology. The implementation of a threshold is widely used to classify using a binary choice, or by the use of multiple thresholds, a range of classifications for an object.

For the detection of a God Component, two different thresholds are introduced. One that is based on the system itself and one that is derived from a benchmark. The system benchmark helps in limiting the number of false positives compared to a generally applicable threshold. It is adaptive to the system itself and can account for the size of the system for which the God Component detection method is applied.
The threshold derived from the system characteristics, however, induces one potential risk. The system will also produce a system based threshold, which may lead to a situation in which the system is ideally in balance and does not contain excessively large component. Due to the nature of the threshold derivation process, one that splits the components somewhere between 0 and 100 percent, at least one component will be classified as a God Component.
To overcome this risk, the introduction of a benchmark threshold is proposed. The usage of a threshold derived from a representative data set containing multiple systems gives an objective lower limit for detection. The maximum of the two thresholds will be used a the threshold against which the components are tested.

Adaptive detection method
Starting from the Automatic Metrics Threshold Derivation method developed by Arcelli et al. [12], the three topics above are processed into an adaptive detection method to detect God Components. The detection method consists of five main steps. The steps are depicted in Figure 4.1 with their respective sub-steps.

The main processes in the method are

1. **Metric computation**
   The first step consists of the collection of the metrics.
   The user must choose a level of granularity to execute the detection on. Detection can take place on class and package level. All components on the chosen level are to be stored in a database to provide the detection method later on with input for the benchmark and the list of components which will be tested against the threshold. For each of the collected components, the number of

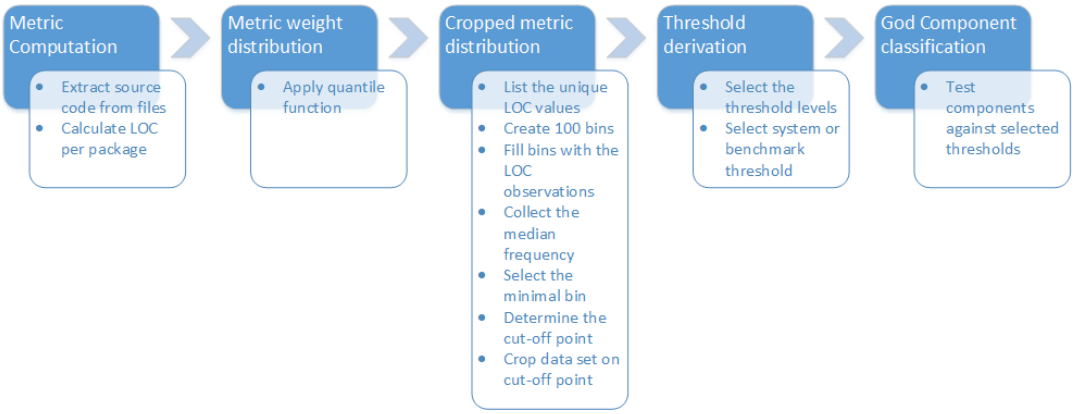

FIGURE 4.1: Process steps for the Adaptive Detection Method

lines of code is registered. The same metric calculation must be applied to all the components to prevent uneven comparison in later steps.

2. **Metric weight distribution extraction**
From the metric set obtained in step one, a distribution plot is created. The data is transformed using the appliance of the quantile function. The creation of a quantile distribution helps in creating the distinction between the lower, common, values and the more excessive values which make up the top of the quantiles.
When examining, the distribution derived with the quantile function shows a heavily skewed distribution. Figure 4.2 depicts this skewed distribution. The lowest values are often close to the lowest possible value of 1. The distribution tends to grow rapidly as it approaches the higher quantiles.

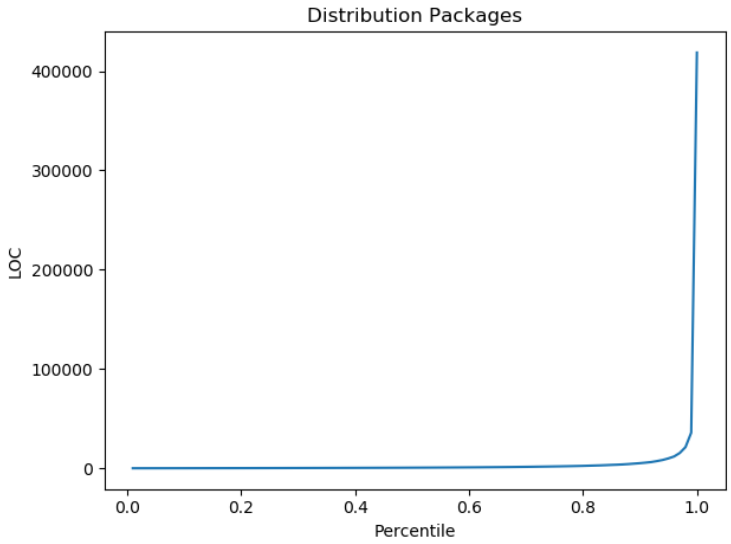

FIGURE 4.2: Quantile distribution for the Qualitas Corpus data set

For the calculation of the benchmark threshold, this step includes the aggregation of all components into a single data set. For each system in the benchmark

the components and its corresponding metric value is added to the distribution.

3. **Cropped metric distribution**
   The third step is concerned with the adaption of the input data into a cropped data set. When applying traditional descriptive statistical metrics, the change arises that the input data is not processed according to its highly skewed character. The cropping of the data should split the input into two sets for which the cut-point lies on the splitting point between acceptable values of the metric and excessive values.

   The specific process for this cropping and the determination of a cut-point is described in detail in the next section.

4. **Threshold derivation**
   Using the cropped data set from the previous step, the selection of the thresholds can be executed. Three levels of thresholds are defined, which corresponds to different levels of impact and urgency to deal with the God Components. These levels are Low, Medium and High and are extracted from the cropped data set by using the 25^{th} percentile for the Low, 50^{th} percentile for the Medium and 75^{th} percentile for the High.

   Using the collected system threshold and performing the same process on the data from the benchmark, a final choice between the two has to be made. The maximum of the two will be the threshold against which the components are tested, namely $max(Threshold_{system}, Threshold_{benchmark})$. If the benchmark provides the lower threshold, the system is expected to be not comparable to the benchmark and should use its own threshold. If the system threshold is the lower value than the benchmark shows that the system threshold is already a value that could be considered acceptable in the industry.

5. **God Component classification**
   The final step is testing all components from step one against the threshold set in the previous step. Each component for which the registered metric is higher than that of the threshold set for the system will be marked as a God Component. The results containing all God Components will be made available to the user. The results contain information on the component identifier, such as the components name, a possible location in the code, and the lines of code for the component.

4.3 Cropped metric distribution algorithm

To effectively split the original data set of lines of code per package into two sets, a cut-off point has to be determined. The splitting of the data set into two sets, one with typical LOC values, and one set with the more extremer LOC values. This split provides a way to determine God Component detection thresholds from the second data set. To select the cut-off point, seven steps are executed. The steps are based on the method by Arcelli et al., which they use for the selection of a benchmark-based threshold [12]. The algorithm in this thesis is adapted to suit the more substantial variation in observations better. The steps below are sub-steps of the cropped metric distribution from the previous section, as also shown in Figure 4.1

The seven steps are the following:

1. **List unique metric values**
   The first step consists of creating a list with all the unique values from the list of Lines of Code per package. All duplicate values in the metric set are removed. The remaining unique values function as a starting point for determining which values are frequent in the whole metric data set and which values are outliers or even extreme values.

2. **Create 100 bins for histogram**
   From the list containing the unique values for the Lines of Code values present in the analyzed data set, a total of hundred bins are formed. The bins will have a variable bin width, which is derived from the percentiles in the unique values list. The percentiles are evenly distributed in the range [1, 100], resulting in an array *a* with 100 elements.
   If *n* is the number of observation in the data set containing all LOC values, *i* be the evenly distributed percentiles, or bins in this equation, than c_i will be the function for the binning of values. The formula is expressed in equation 4.1.

$$n = \sum_{i=1}^{100} c_i \tag{4.1}$$

   The function c_i represent the counting of values for each bin. The count represents all values that are higher than the lower limit of the bin and are lower than or equal to the upper limit of the bin. p_i is the percentile function in which *i* is the percentile used. x_j represent all values in LOC metrics data set.

$$c_i = \sum_{j} 1_{(p_{i-1}, p_i](x_j)} \tag{4.2}$$

   The result of this step are a hundred bins which map to the hundred percentiles of the unique value list.

3. **Fill bins with LOC observations**
   The third step calculates the frequency table for the hundred bins created in the previous step. Each value in the original data set is placed inside a bin. For each bin, the count of values in it is registered in the frequency table. The outcome is an overview of the number of observations placed in each bin. This result can be used to extract the cut-off point further on.

4. **Collect frequency belonging to the median bin**
   In the frequency table, the median bin is selected, and its frequency count is selected to be used as the input for the search of the cut-off value.

5. **Select minimal bin with frequency value**
   The fifth step consists of selecting the bin, which will determine the cut-off point in the original data set. Using the frequency value from the previous step, the lowest bin is selected for which all frequency values in higher bins are equal to or lower than the frequency value from the previous step.
   The step creates a distinction between the more common values in the system and the point after which the observations trend is strictly downwards.

6. **Lookup lower limit from the minimal bin in full data set**
   The cut-off point in the original data set will be the value from the selected bin in the previous step. The lower limit of this bin will function as the value on which the data set will be cut-off.

7. **Split data set on cut-off point**
   The original data set, namely the components in the system under analysis, is sorted on the Lines of Code values from low to high to determine the index of the cut-off value. The first occurrence of a value equal to or greater than the value from the previous sub-step is selected. The index of this first occurrence is then used to split the set in two where the cropped metric set contains all values from the derived index up to and including the highest value in the sorted set.

This algorithm mainly differs from the one developed by Arcelli et al. in one crucial step: the extraction of the frequency table. The algorithm from Arcelli et al. extracts a hundred quantiles from the total distribution. Following that, they count the number of occurrences for each of the extracted percentile value. This method creates the limitation that a small fluctuation in the distribution may result in an undesired low or high cut-off point.
As an example to demonstrate such case, let us suppose that the selected median in the frequency table has a frequency of 1. If by chance the 98^{th} percentile has a count of 2, the selected cut-off point will be based on the first bin count that is again the minimum for which all following bin counts are equal or lower than the median value: in this example, the 99^{th} percentile. Table 4.1 contains an example of 19 quantiles. The median, bin n. 10, is printed in bold. Using the selection method, the first bin for which all the following bins would be equal to or lower than the value of the median is the nineteenth bin. This outlier, however, disturbs the selection of the minimal bin, as the selection method could be applied from the fifth bin if the eighteenth bin does not show a significantly higher value than the rest of low-value bins. In comparison with the total number of observations, the difference between the frequency in the fifth and eighteenth bin in neglectable.

Bin	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16	17	18	19
Value	10	21	34	50	65	80	100	150	200	**275**	380	500	650	825	1,100	1,350	1,600	1,950	2,450
Frequency	1,581	1,240	150	12	1	1	1	1	1	**1**	1	1	1	1	1	1	1	2	1

TABLE 4.1: Frequency distribution example

By using the percentiles as bin limits, this effect is elided. Furthermore, it creates a more accurate view of the distribution of the whole metric data set.

4.4 Benchmark threshold

The threshold for classifying a component as a God Component in the proposed method relies on the maximum of two values. The value of the system itself can be extruded by applying the method on the system. The threshold for the benchmark set can be collected by applying the method on a set of benchmark systems. In this section, an example calculation for the benchmark threshold is presented. It also functions as the benchmark threshold for the implementation in Arcan.
As Arcan is a tool for analyzing Java systems for the presence of architectural smells, the benchmark data set chosen is thus a collection of Java systems. Tempero et al. provide a curated collection of Java systems which are suited for empirical studies, known as the Qualitas Corpus (QC) [23]. This collection of Java systems consists of 109 open source systems from multiple fields, including graphical software, parsers, a database, and testing frameworks. Table 4.2 contains the main metrics of the Qualitas Corpus. For this research, version 20130901 of the Qualitas Corpus is used.

Metric	Value
Number of systems	109
Number of packages	15,821
Number of classes	191,466
Lines of Code	23,002,894

TABLE 4.2: Metrics of the Qualitas Corpus, version 20130901

For Arcan, the implementation will focus on package level. Therefore, the adaptive detection method is followed using the Qualitas Corpus as input. Below are all the steps described in detail.

1. **Metric computation**
   The first step is the collection of the metrics. For each system in the Qualitas Corpus, a metadata file is available which lists all classes, including their package and number of Lines of Code. From this information, the total number of Lines of Code per package can be calculated using the unique values in the package name list and sum the Lines of Code that are part of the respective package. The metadata file contains metrics for the Lines of Code including blank and commented lines and values for only non-blank, non-commented Lines of Code. The latter will be used for the reason stated earlier in section 4.2 The result is a list with package names and the number of Lines of Code that are included in that package.

2. **Calculate weight distribution**
   The second step is calculating the weight distribution for the collected Lines of Code per package. The packages and their LOC metric are aggregated into a single data set. The result is a set with around 15k of entries. Figure 4.3 depicts the distribution of LOC in the benchmark set based on a hundred quantiles of the whole distribution of LOC. As depicted, the distribution is highly left-skewed. The majority of the values are relatively together with a small percentage of all values growing rapidly at the end.

3. **Collect cropped distribution**
   The third step is the cropping of the distribution into a set with the extreme values of the original data set. The result should be a cropped data set from

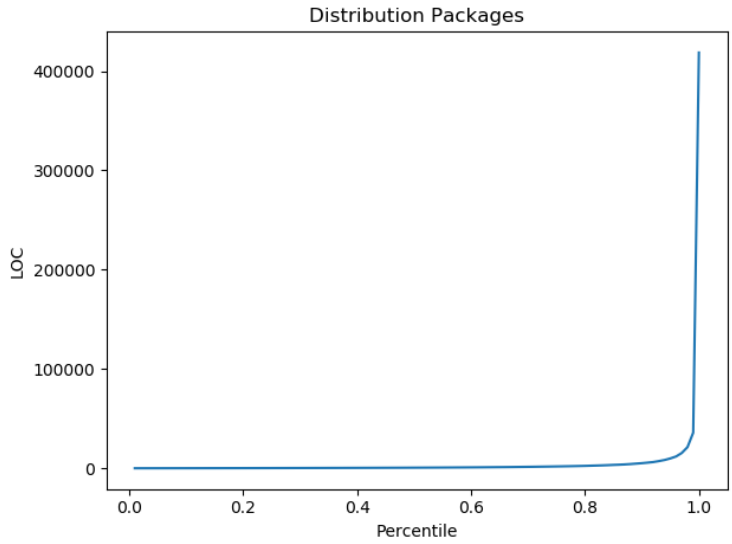

FIGURE 4.3: Distribution LOC per package for the Qualitas Corpus data set

which the thresholds can be extracted. The cropping is done using seven sub-steps, beginning with

(a) *List unique metric values*
From the total set of LOC values, all unique values are added to another data set. A total of 4,006 unique values remain. In Table 4.3 some descriptive statistics can be found.

Observations	**4,006**
Minimum LOC	6
25^{th} quantile	1,009
50^{th} quantile	2,188
75^{th} quantile	4,483
Maximum LOC	18,6835

TABLE 4.3: Unique values benchmark

(b) *Create 100 bins per histogram*
Using the unique value list from the previous sub-step a total of hundred bins are created. The lower limit of bin_i is greater than percentile(a_{i-1}) in the unique value list. The upper limit of bin_i is lower or equal to percentile(a_i) of the unique value list with a range for *i* of [1,100]
The bin width varies between the 40 and 15,453 Lines of Code.

(c) *Fill bins with LOC observations*
The next sub-step is the creation of a histogram for the defined bins. All LOC values of the packages are placed in the correct bin. The extracted distribution can be found in Figure 4.4

The x-axis, containing the number of LOC per package, is plotted on Log-scale to improve the visualization of the graph. It provides a clear image

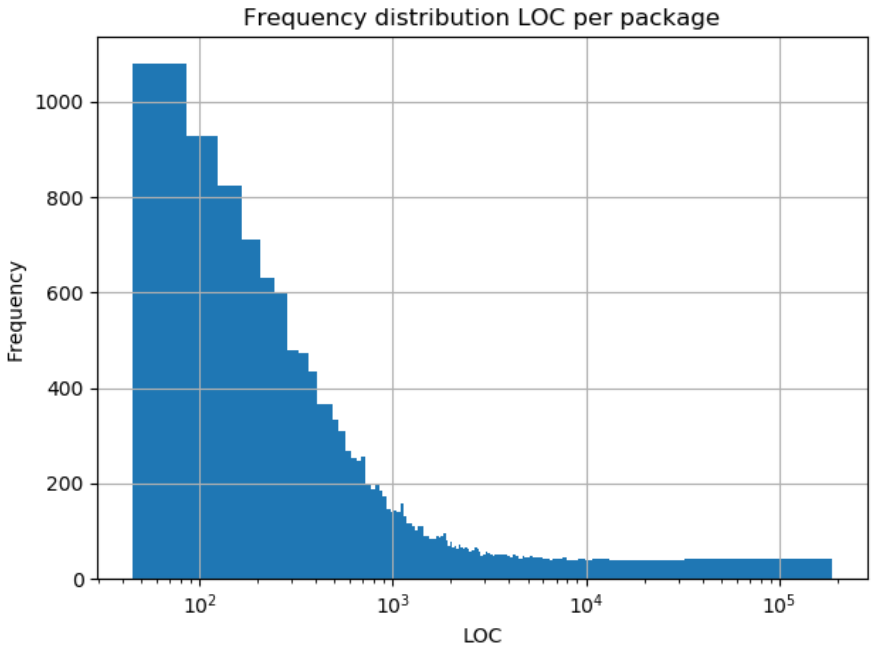

FIGURE 4.4: Histogram LOC per package for the Qualitas Corpus data set

on the fluctuation in the number of observations in a bin for the higher LOC values.

(d) *Collect frequency belonging to the median bin*
The median bin has a total of 65 observations that fall into the bin.

(e) *Select minimal bin with frequency value*
When searching for a bin for which all bins have a frequency equal or less than that of the median bins the 55^{th} bin is the first to fulfill this requirement. The lower limit of this bin is a total of 2,491 Lines of Code in a package.

(f) *Lookup lower limit from the minimal bin in full data set*
The lower limit value of 2,491 LOC will be used as the value to cut-off the original data set. The first occurrence of this value in the complete set of LOC per package is at index 13,703. The cropped data set will be the set of values from index 13,703 up to the maximum index.

(g) *Split data set on cut-off point*
The cut-off line lies at roughly the 87^{th} percentile of the Qualitas Corpus data set. The cropped set will, therefore, consist of the top 13 percent of the whole data set. The cropped data set contains a total of 2,118 observations of Lines of Code per package.

4. **Threshold derivation**
   The final step is to select the thresholds from the cropped data set. Looking up the 25^{th}, 50^{th}, and 75^{th} quantile the limits for the Low, Medium, and High threshold are calculated. Table 4.4 show the respective value for each threshold.

Threshold	Quantile	LOC value
Low	25^{th}	3,455
Medium	50^{th}	4,727
High	75^{th}	7,593

TABLE 4.4: Threshold derived from Qualitas Corpus

Lanza and Marinescu also use the symbolic naming of threshold using Low, Medium, and High categorization [15]. For the Lines of Code metric, they use their High threshold. This research will report for each threshold the number of God Components detected and report the confidence level back to the users using the Low, Medium, and High classifications. However, the Medium and High classification are advised to be used. These classifications provide a balance between selecting the extreme values as well as detecting all God Components.

Chapter 5

Validation

In this chapter, the developed adaptive detection method and the implementation of the method in Arcan are validated. The first section contains the setup of the validation. The section that follows presents the results of the validation based upon multiple Java systems taken from the Qualitas Corpus.

5.1 Validation set-up

Using a selection of systems available in the Qualitas Corpus, the effectiveness of the adaptive detection method is tested. The set of systems is analyzed by Arcan to also investigate the effectiveness of the implementation. The systems are selected based on one or more of the following characteristics:

- The system consists of a large number of packages.
- The system is part of the test systems used in the development of Arcan.
- The system is one of the larger sized projects in the Qualitas Corpus in terms of LOC.
- The system is a small project.

The selected systems that are evaluated are:

- Ant: a software tool to automate software build processes. It was selected due to its small size, so the impact of the detection method on a small systems can be determined.
- Azureus, or Vuze: a BitTorrent client used for transferring files using the BitTorrent protocol. It is included in the test system for its larger number of packages.
- Derby: a relational database management system. Derby is previously used in the evaluation of the detection capability for other architectural smells in Arcan.
- Spring framework: an application framework used for the modeling and configuration of Java systems. The framework is previously used in the evaluation of the detection capability of other architectural smells in Arcan.

These points, and the previous ones are meant to increase the external validity of the detection algorithm. The key metrics of the systems mentioned above are contained in Table 5.1.

Project	version	category	Number of Packages (NOP)	Lines of Code (LOC)
Ant	1.8.4	Generator/parser	70	109,037
Azureus	4.8.1.2	BitTorrent client	480	489,283
Derby	10.6.1.0	Database manager	110	284,163
Spring	3.0.5	Framework	281	190,369

TABLE 5.1: Analyzed projects

The validation is done in three steps for each of the system analyzed:

1. **Run Arcan and use the God Component detection**
   The system is run through Arcan where all dependencies are calculated. Using a set of compiled *.jar* or *.class* as input, a graph database is constructed. As Arcan analyses bytecode, the results may differ from that of the original source code. The Arcan runs are therefore also a validation on the decompiler used.
   The Lines of Code metric calculations are performed on the reconstructed source files. Following that, the package metric calculator calculates the sum of the Lines of Code for each package based on all classes which are part of the package. For this, the graph is scanned for class nodes that are connected to the package with a class-package dependency edge. The next step is the extraction of the system threshold. Based on the Lines of Code metric, each package is tested against the selected threshold. Packages that are equal or higher than the threshold are flagged as a God Component with their confidence levels (i.e., High, Medium, Low, based on the identified thresholds).

2. **Detection of false positives**
   To detect false positives, the original source code is compared to that of the reconstructed files by Arcan. The Qualitas corpus provides metadata for each system, containing the number of Lines of Code and the package name for each class. The sum of Lines of Code is calculated for each package and compared to that of the metric as calculated by Arcan. The Lines of Code metric is saved along with the other metadata and metrics that Arcan computes on the dependency graph so it is available for future use.
   Packages from the source code whose Lines of Code do not exceed the threshold but are flagged by Arcan as a God Component are considered as False Positives. For the threshold, the High threshold is used to test for false positives as it has the greatest impact on the system.

3. **Detection of false negative**
   The detection of false negatives is done by the comparison between the source code and the reconstructed source. For both, the Lines of Code per package are extracted and compared to see if any packages in the source exceed the threshold but are not flagged in Arcan as being a God Component. Furthermore, for each system analyzed, a box plot distribution is plotted. This plot is used to visually check if the system shows any outliers that might have been missed by the Arcan tool. This visualization could indicate large gaps between clusters of Lines of Code metrics, possibly indicating the presence of a God Component. For the threshold, the High threshold is used to test for false positives as it has the greatest impact on the system.

The results of the above-mentioned approaches are described in the next section.

5.2 Validation results

Threshold derivation

The results of the threshold derivation process in Arcan for each project are listed in Table 5.2. For each system, the total number of Lines of Code is listed. Furthermore, the calculated Low, Medium, and High system threshold are presented. The final column contains the used threshold based on the maximum between the benchmark thresholds, as in Table 5.3, and the system thresholds.

		System Treshold			
Projects	**LOC**	**Low**	**Medium**	**High**	**Used Threshold**
Ant	109,037	533	916	2,113	Benchmark
Azureus	489,283	703	1,199	2.324	Benchmark
Derby	284,163	4,213	13,433	16,444	System
Spring	190,369	1,613	2,212	2,664	Benchmark

TABLE 5.2: Threshold results for analyzed projects

Threshold	**LOC value**
LOW	3,455
MEDIUM	4,727
HIGH	7,593

TABLE 5.3: Benchmark thresholds derived from Qualitas Corpus

For all systems analyzed, only Derby provides a system benchmark, for which the thresholds are higher than the ones derived from the benchmark. The main reason for this system-used threshold is that Derby contains a small number of packages, but the Lines of Code per package are higher than generally seen in the other systems. In comparison with the other systems, packages in Derby contain more Lines of Code. This characteristic makes that the cut-off point is already above the Low threshold of the benchmark. This high cut-off point further influences the selection of the system thresholds, which are in the higher number of Lines of Code.

God Component detection

Using the derived thresholds, each system is analyzed to detect if there are any God Components. The results of these detection runs are listed in Table 5.4. For each system, the number of packages (NOP) is listed in combination with the number of detected God Components for the Low, Medium, and High thresholds. For each threshold mentioned, all packages are counted that are equal to or greater than the used threshold. The Low threshold, therefore, includes the God Components detected in the Medium and High threshold runs. The Medium threshold also includes the results of the High threshold runs. The final column shows which thresholds, the system thresholds, or the benchmark thresholds, are applied.

Projects	NOP	God Components			Used Threshold
		Low	Medium	High	
Ant	70	9	4	2	Benchmark
Azureus	480	30	17	8	Benchmark
Derby	110	12	8	4	System
Spring	281	4	2	0	Benchmark

TABLE 5.4: God Component detection results for analyzed projects

For each system, Arcan detected God Components for the Low and Medium level thresholds. Only for the Spring system, the High threshold did not return any God Components. The low maximum Lines of Code explain this exception for the packages in Spring. The highest found package in Arcan contained only 6,987 Lines of Code, which is below the benchmark threshold of 7,593.
The detected God Components in the Derby systems are increasing with the same distance for the different thresholds. This distance is explained by the usage of the system threshold, which makes use of the 25^{th}, 50^{th}, and 75^{th} percentile, resulting in an even distance between values from the cropped data set.

Although Arcan provides all three threshold level to the user, the user is advised to address and prioritize Medium and High classified God Components, as such classification levels provide the most solid detection for God Components. The Low threshold level, instead, tends to start detecting packages that are not significantly larger than their neighbors and have a high chance of being false positives God Components.

Precision and recall

To determine the effectiveness of Arcan, the results presented in Table 5.4 are evaluated to detect any false positives and false negatives. The table shows the results for the High threshold, as we consider it the level with the largest impact on the system. Using the source code available in the Qualitas Corpus, the packages are scanned for their original Lines of Code and compared to the Lines of Code as calculated by Arcan. The findings of this test are reported in Table 5.5. Furthermore, the source code distribution is plotted using a box plot to perform a visual validation that no possible God Components are missed against the High threshold. The plots are depicted in Figure 5.1.

	Ant	Azureus	Derby	Spring
Total God Components	4	7	6	3
True Positive	2	7	3	0
False Positive	0	1	1	0
False Negative	2	0	3	3
True Negative	0	0	0	0
Precision	100%	87.5%	75%	0%
Recall	50%	100%	50%	0%
F-measure	66.67%	93.32%	60%	0%

TABLE 5.5: Confusion matrix for analyzed projects using the High threshold

Arcan is able to operate with a precision rate between 75% and 100% for the God Component detection. This precision rate is slightly lower than the precision rate of 100% when detecting for Cyclic Dependency, Unstable Dependency, and Hub-Like Dependency reported in earlier work by Arcelli et al. [10]. A side-note, however, must be made on those results, as it is limited in the number of projects and the small size of the data set analyzed. The number of false positives is limited to a single package in some of the systems. The cause of these false positives are the Lines of Code in the reconstructed source code. The Lines of Code provided by Arcan are around 10% higher than the number of Lines of Code in the source, making the packages just above the used threshold. Future work into the reliability and precision of the reconstructor could help to overcome this issue.
The recall is lower than that of the precision rate. Here also the reconstructor plays a role. Source code and reconstructed code do differ in the number of Lines of Code. This difference results in undetected God Components by not having the correct amount of Lines of Code available. A possible explanation for the lower Lines of Code in Arcan is that the reconstructor has less clutter in the reconstructed code than the original source code. The false negatives may, therefore, become indicators of fuzzy declarations in the source code. This possibility is, however, out of scope for this research, and not further examined.

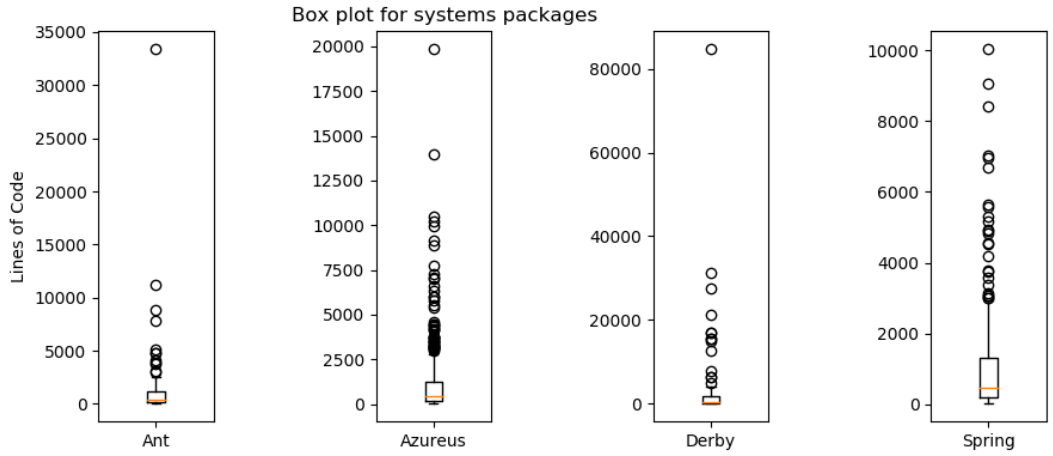

FIGURE 5.1: Boxplot distribution for the systems

The box plots in Figure 5.1 confirm the results of the manual check on the source files. The combined results of the true positives and the false negatives cover the definition of the God Component. The total God Components mentioned in Table 5.5 correspond with outliers who broken loose from the outliers connected to the inter-quantile distance.
The adaptive detection method itself seems therefore suitable to detect God Components based on either the system thresholds or the benchmark thresholds.

Chapter 6

Conclusion and future work

In this chapter, we conclude the thesis with a section containing the conclusions drawn during the research project, after which the thesis ends with a section containing recommendations for future work.

6.1 Conclusion

In chapter 3 the research goal was formulated as:

Develop and validate a detection method for the God Component smell that takes into consideration the size of the project itself and implements this method in the Arcan tool.

The goal was based on the absence of a context-aware detection method for the God Component smell in literature. This thesis proposes a new definition for a God Component:

A component whose size exceeds by far the average size (i.e., is an outlier) of all components in the system or the average size from the industry benchmark.

The proposed definition takes into consideration the size of the component with respect to other components in the systems. It also introduces the usage of a benchmark to deal with the increasing size of software projects and therefore the shifting standard of an acceptable size for components in software systems.
From the definition, an adaptive detection method is created, which deals with the differences between components. The method contains steps to ensure that the detection of God Components is based on thresholds that consider the system it is analyzing. By analyzing the number of Lines of Code per package, the method can compare the size between different packages. A cropping algorithm separates the outliers from the average values, after which system threshold can be extracted. The same algorithm is applied to a benchmark set of Java systems to provide a solid foundation of the minimal size of what classifies a God Component.

The adaptive detection method is implemented in Arcan, an Architectural Smell tool, to evaluate the effectiveness of the detection method as well as to provide a practical application of the detection method. Arcan uses bytecode to analyze the systems, which require a code reconstructor to extract the number of Lines of Code for the components. This addition also creates possibilities to use the LOC metric for the detection of other smells.

The development of the algorithm and the implementation contribute to the detection of God Components that are not based on a static threshold but instead on the system derived threshold or the benchmark threshold. This development fulfills the first part of the research goal.

To validate the correctness of the adaptive detection method, four systems are tested for God Components using Arcan. The result of these runs can be found in Table 5.5. Arcan shows for all system a sufficient precision in detecting God Components. The recall rate still requires improvement, but the error present explains the cause for these lower rates in the reconstruction of the Lines of Code. Using the box plots for the different systems, the adaptive detection method was validated without the side-effect of the tool in which it is implemented. The results of this comparison show that the adaptive detection method itself can detect all God Components as defined by the proposed God Component definition at the beginning of chapter 4.

Providing the above arguments, we conclude that the developed adaptive detection method is able to detect God Component using variable thresholds successfully. The thesis offers a substantial report on the new God Component definition and how it can be detected. The presented work adds to the existing literature on Smells detection, and specifically on the God Component smell detection.

6.2 Future work

The adaptive detection method is suitable for the detection of God Components in Java systems. However, several limitations could function as the start for future work. With the research performed in this thesis, the following topics could be investigated in future work:

- **Variable number of bins**
  The current cropping algorithm implemented in the detection method uses percentiles for binning and selecting a cut-off point. Although this implementation can be applied to smaller systems, it could reduce the validity of the selected system's thresholds. The using of percentiles on systems that only contain unique values would ignore the distribution of the Lines of Code per package.
  The usage of a variable number of bins could reduce the impact of the situation, as mentioned above. Future work on this could investigate the impact on the current method as well as the correctness of detection in smaller systems.

- **God Component packages analysis**
  The proposed adaptive detection method currently only uses the size of the package and test this against the derived threshold. Characteristics like the goal of the package are not taken into consideration. One could argue that certain types of packages are likely to be significant in size due to their nature. Parsers are an example of this. Also, the detection method could be extended with separating core packages from supporting packages to further support the developers in addressing possible technical debt in their systems.

- **Java source code reconstructor**
  As already addressed in chapter 5, the false positives and false negatives are

largely due to the difference in Lines of Code between the source and the reconstructed code from Arcan. A more precise reconstructor could improve the reliability of the detection and make the LOC metric suitable for other detection methods as well. A study on the difference between the source and reconstructed code can indicate the challenges as well as opportunities. The reconstructed code could provide insight into how to write methods more effectively using less code.

Bibliography

[1] Tiago L Alves, Christiaan Ypma, and Joost Visser. "Deriving metric thresholds from benchmark data". In: *2010 IEEE International Conference on Software Maintenance*. IEEE. 2010, pp. 1–10.

[2] Paris Avgeriou et al. "Managing technical debt in software engineering (dagstuhl seminar 16162)". In: *Dagstuhl Reports*. Vol. 6. 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.

[3] Umberto Azadi, Francesca Arcelli Fontana, and Davide Taibi. "Architectural Smells Detected by Tools: a Catalogue Proposal". In: *International Conference on Technical Debt (TechDebt 2019)*. 2019.

[4] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[5] F Buschmann et al. *Software Patterns*. John Wiley & Sons, 1996.

[6] Shyam R Chidamber and Chris F Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on software engineering* 20.6 (1994), pp. 476–493.

[7] Ward Cunningham. "The WyCash portfolio management system". In: *ACM SIGPLAN OOPS Messenger* 4.2 (1993), pp. 29–30.

[8] Karin Erni and Claus Lewerentz. "Applying design-metrics to object-oriented frameworks". In: *Proceedings of the 3rd international software metrics symposium*. IEEE. 1996, pp. 64–74.

[9] Eduardo Figueiredo et al. "On the impact of crosscutting concern projection on code measurement". In: *Proceedings of the tenth international conference on Aspect-oriented software development*. ACM. 2011, pp. 81–92.

[10] Francesca Arcelli Fontana et al. "Arcan: a tool for architectural smells detection". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE. 2017, pp. 282–285.

[11] Francesca Arcelli Fontana et al. "Automatic detection of instability architectural smells". In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2016, pp. 433–437.

[12] Francesca Arcelli Fontana et al. "Automatic metric thresholds derivation for code smell detection". In: *Proceedings of the Sixth international workshop on emerging trends in software metrics*. IEEE Press. 2015, pp. 44–53.

[13] Joshua Garcia et al. "Identifying architectural bad smells". In: *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE. 2009, pp. 255–258.

[14] Neil B Harrison, Paris Avgeriou, and Uwe Zdun. "Using patterns to capture architectural decisions". In: *IEEE software* 24.4 (2007), pp. 38–45.

[15] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[16] Duc Le et al. "Toward a Classification Framework for Software Architectural Smells". In: *Techincal Report csse. usc. edu* (2017).

[17] Martin Lippert and Stephen Roock. *Refactoring in large software projects: performing complex restructurings successfully*. John Wiley & Sons, 2006.

[18] Isela Macia et al. "Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems". In: *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM. 2012, pp. 167–178.

[19] Md Abdullah Al Mamun, Christian Berger, and Jörgen Hansson. "Correlations of software code metrics: an empirical study". In: *Proceedings of the 27th international workshop on software measurement and 12th international conference on software process and product measurement*. ACM. 2017, pp. 255–266.

[20] Robert Martin. "OO design quality metrics". In: *An analysis of dependencies* 12 (1994), pp. 151–170.

[21] Thomas J McCabe. "A complexity measure". In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[22] Raed Shatnawi et al. "Finding software metrics threshold values using ROC curves". In: *Journal of software maintenance and evolution: Research and practice* 22.1 (2010), pp. 1–16.

[23] Ewan Tempero et al. "Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies". In: *2010 Asia Pacific Software Engineering Conference (APSEC2010)*. Dec. 2010, pp. 336–345. DOI: http://dx.doi.org/10.1109/APSEC.2010.46.

[24] Marc Van Kempen et al. "Towards proving preservation of behaviour of refactoring of UML models". In: *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*. South African Institute for Computer Scientists and Information Technologists. 2005, pp. 252–259.