Bachelor Informatica

University of Amsterdam

# The effect of modularity on software quality

Stefan van der Woude

June 8, 2018

**Supervisor(s):** dr. A.M. Oprescu

**Signed:**

**Abstract**

Modular programming is a highly used software design pattern in both private and open source software projects. Despite previous research on the effect of modularity on the development cycle, there is no recent research on the effect of modularity on the quality of software using the ISO/IEC 25010 standard. In this work, a method is presented to detect the level of modularity for any software project. This method is an alternative to existing methods, providing a simple but effective way of defining the modularity of a program. By using this method, the modularity of a program can be compared to various quality characteristics. We performed several sets of experiments to analyze the effect of modularity on the quality of software, as defined in the ISO/IEC 25010 standard. Our analysis shows that there is in fact a positive correlation between the modularity and the quality of Python programs. We also find a weak correlation between modularity and quality of Java programs.

# Contents

CHAPTER 1

# Introduction

Creating high quality software is often very difficult since you want to keep your solution short and simple, or face a problem that is too complex to be solved without using bad practices. Writing high quality software becomes even harder when the size of the code base increases, since more code leads to a higher complexity [1], in turn making it harder to maintain the code base [2]. Over the years, many standards and guidelines have been developed to provide general definitions and rules for software. Examples of this are the Don't Repeat Yourself (DRY) concept [3], cyclomatic complexity metric [4] and test coverage measurements [5]. Even though there are in many cases clear quantitative guidelines for these values and concepts, they are hard to strictly follow in complex programs.

If we look at a more general view of software, we get to the field of software design. Software design is the process by which an agent creates a specification of a software artifact, intended to accomplish goals, using a set of primitive components and subject to constraints [6]. One of these design techniques is modular programming [7], which we will analyze in this thesis. Modular programming emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.

## 1.1 Research questions and methodology

The goal of this research is to gauge whether there is a relation between the degree of modularity of a program and the quality of that program. Moreover, we study the effect of modularity on different quality characteristics: maintainability and performance efficiency. The research question that will be answered is as follows:

"What is the effect of modularity on the quality of software?"

In order to answer this question, a few other questions have to be answered. Since we have to define the degree of modularity of programs, we will have to answer the question "How can we easily and effectively calculate the degree of modularity of a program?". After this question has been answered, we have to find methods for measuring all (sub-) quality characteristics used to define the quality of software. Thus, for each characteristic we have to find an answer to the question "How can we measure [quality attribute] of a program?". Based on the answers to these questions we build a framework to perform measurements on programs; The results are used to answer the main research question.

This research uses a quantitative approach. Experiments are done using existing literature and new ideas in order to generate data which can be used for statistical analysis. Using this analysis we will be able to analyze to what degree there is a relation between the modularity and quality of software. In this work a method of measuring the degree of modularity of a program is

developed, after which a set of programs is tested on different quality characteristics, including modularity.

## 1.2 Contributions

This work will expand on existing research on software measurements regarding modularity [8–10] of Python and Java programs. The outcome of this research will provide a new simple and effective method of measuring modularity. Besides providing this new methodology, a framework is created with which the quality characteristics maintainability and performance efficiency can be measured in any Python program. We will create new metrics built on top of existing metrics [11] to allow for measurements on these characteristics.

## 1.3 Thesis outline

In order to answer our research question, we will have to go through a few steps. First of all, we will look into some basic information and concepts that are used throughout this work. After information about software quality and software design, we look at the experiments that are necessary to answer our research question. In chapter 4 we will look at the experimental setup that is used to perform the experiments, after which the results of all experiments are shown in chapter 5. Finally we will discuss these results and come to a conclusion in chapter 8.

CHAPTER 2

# Background

Writing good code is something you strive for from the day you start learning programming. One aspect of high quality software is maintainability [12,13]. In turn, code quality improves the maintainability of a program, for example by lowering the complexity of the program [2]. Besides a low complexity, one could think that the lack of bad practices in your code base could also be an indication of maintainable software. However, research has shown that these code smells do not necessarily have a direct negative (nor a positive) effect on maintainability [14]. It is shown that code conventions, sets of guidelines for programming languages that recommend programming style, improve the readability of the software by applying some level of consistency which can be analyzed faster [15]. These conventions range from naming conventions to architectural best practices.

## 2.1 Software quality

In 2011, the International Organization for Standardization released the ISO/IEC 25010 - 'System and software quality models' standard [13]. This standard defines a set of models and characteristics to test software quality. This standard is a successor to the ISO/IEC 9126 - 'Product quality - Quality model' standard [12] released in 2001. The latest version defines these characteristics in more detail and provides a new categorization of all characteristics and their sub-characteristics.

### 2.1.1 ISO/IEC 25010

The ISO/IEC 25010 standard defines eight characteristics that can be evaluated when looking at the quality of a piece of software. Some of these relate to the outcome of interaction when a product is used in a particular context, others relate to the static properties of software and dynamic properties of the computer system it is used in. We proceed to analyze the full list of characteristics in alphabetical order from the perspective of our research.

### 2.1.2 Compatibility

The first quality characteristic is compatibility. Compatibility is referred to as the degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment. ISO divides this into two sub-characteristics: co-existence and interoperability.

#### Co-existence

The first sub-characteristic is co-existence. Co-existence is the degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products without detrimental impact on any other product.

#### Interoperability
The interoperability of two or more systems, products or component defines the degree to which they can exchange information and use the information that can be exchanged.

### 2.1.3 Functional suitability

The second quality characteristic is functional suitability, this characteristic defines the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions. This characteristic consists of three sub-characteristics: functional appropriateness, functional completeness and functional correctness.

#### Functional appropriateness
The functional appropriateness of a system defines the degree to which the set of functions facilitate the accomplishment of specified tasks and objectives.

#### Functional completeness
The second sub-characteristic defines the degree to which the set of functions in a system covers all the specified tasks and user objectives. We call this the functional completeness of a product or system.

#### Functional correctness
The functional completeness of a product or system defines the degree to which it provides the correct results with the needed degree of precision.

### 2.1.4 Maintainability

The next characteristic, after functional suitability, is maintainability. Whilst maintainability could be evaluated differently per person, there are some aspects of a program that can be measured to determine the maintainability of a program. In this case we refer to maintainability as the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it, or adapt to changes in environment, and in requirements. This characteristic consists of four aspects.

#### Analysability
Analysability is the degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.

#### Modifiability
The second sub-characteristic, modifiability, is referred to as the degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.

#### Modularity
Whilst we are looking at the effect of modularity on the quality of software, ISO/IEC 25010 already considers this to have an effect on the maintainability of a product or system.

#### Re-usability
With re-usability, we refer to the degree to which an asset can be used in more than one system, or in building other assets.

#### Testability
The final sub-characteristic of maintainability is testability. Testability defines the degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

### 2.1.5 Performance efficiency

The fourth characteristic, performance efficiency, presents the performance relative to the amount of resources used under stated conditions. This means that three components of performance and resources have to be tested.

#### Capacity

The first component is capacity, capacity is defined as the degree to which the maximum limits of a product or system parameter meets the requirements.

#### Resource utilization

Next, performance efficiency contains the sub-characteristic resource utilization. With resource utilization we mean the degree to which the amounts and types of resources used by a product or system, when performing its functions, meet the requirements.

#### Time behavior

The final component to look at is the degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet the requirements. We refer to this as the time behavior of a product or system.

### 2.1.6 Portability

The next characteristic defines the degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another. We call this characteristic the portability of a system, product or component. When defining the portability, we can look at three different sub-characteristics: adaptability, installability and replaceability.

#### Adaptability

With adaptability, we refer to the degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operation or usage environments.

#### Installability

The second sub-characteristic defines the degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment. We refer to this as the installability of a product or system.

#### Replaceability

The final sub-characteristic, replaceability, defines the degree to which a product can replace another specified product for the same purpose in the same environment.

### 2.1.7 Reliability

The next quality characteristic is reliability. With reliability we refer to the degree to which a system, product or component performs specified functions under specified conditions for a specified period of time. We can split this into four smaller characteristics.

#### Availability

With availability we refer to the degree to which a system, product or component is operational and accessible when required for use.

#### Fault tolerance

The second characteristic defines the degree to which a system, product or component operates as intended despite the presence of hardware or software faults. We call this characteristic the fault tolerance of a system, product or component.

#### Maturity
The next sub-characteristic is maturity. Maturity is defined as the degree to which a system, product or component meets needs for reliability under normal operation.

#### Recoverability
The final sub-characteristic, recoverability, defines the degree to which, in event of an interruption or failure, a product or system can recover the data directly affected and re-establish the desired state of the system.

### 2.1.8 Security

The seventh quality characteristic defines the degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization. We call this characteristic security. This can in turn be divided into five sub-characteristics such as confidentiality and integrity.

#### Accountability
The first sub-characteristic is accountability. Accountability shows the degree to which the actions of an entity can be traced uniquely to the entity.

#### Authenticity
Next up we define the degree to which the identity of a subject or resource can be proven to be the one claimed. We refer to this as authenticity.

#### Confidentiality
In a product or system, confidentiality is very important. The confidentiality of a product or system defines the degree to which it ensures that data is accessible only to those authorized to have access.

#### Integrity
The degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data is referred to as integrity, which is the fourth sub-characteristic of security.

#### Non-repudiation
The final sub-characteristic is non-repudiation. This characteristic defines the degree to which actions or events can be proven to have taken place, so that events or actions cannot be repudiated later.

### 2.1.9 Usability

The final characteristic in the ISO 25010 standard is usability. Usability defines the degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. We can split this into the six sub-characteristics defined below.

#### Accessibility
The first sub-characteristic is accessibility. With accessibility we define the degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use.

#### Appropriateness recognizability
The degree to which users can recognize whether a product or system is appropriate for their needs is referred to as the appropriateness recognizability of a product or system.

#### Learnability
One of the sub-characteristics of usability is the learnability of a product or system. We define this as the degree to which a product or system can be used by specified users to achieve specified goals or learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use.

#### Operability
The fourth sub-characteristic of usability is operability. The operability of a product or system defines the degree to which it has attributes that make it easy to operate and control.

#### User error protection
The next sub-characteristic defines the degree to which a system protects users against making errors. We refer to this as user error protection.

#### User interface aesthetics
Finally, the sub-characteristic user interface aesthetics defines the degree to which a user interface enables pleasing and satisfying interaction for the user.

## 2.2 Software design

As discussed in chapter 2, achieving high quality software could be done using code conventions. An example of an architectural code convention is the use of a software design pattern. These patterns, or techniques, describe the design of the design: how will the program be structured? Edsger W. Dijkstra referred to this layering of semantic levels as the "radical novelty" of computer programming [16]. As we have seen, these code conventions, such as using a software design pattern, improve the readability (or analysability) of a program. In turn, increasing the analysability of a program increases the maintainability of the program, according to modern standards [12, 13]. In this thesis we focus on the Modular Programming design technique.

### 2.2.1 Modular programming

Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of a desired functionality. Although not all programming languages formally support the concept of modules, languages like C# [17], Java [18] and Python [19] do.

In 1975, Fred Brooks observed that there is a positive correlation between the size of an organization and the delay in time to market a software product, we call this Brooks' Law [20]. Brooks argued that adding more programmers to a software project that is already at a late stage of the development process further delays the project. He reasoned that the complexity and communication cost of a project increase with the square of the number of developers, but the amount of work done only increases linearly [21].

This problem also occurred in the development of the Linux Kernel, which is an open source project [22]. In 1996, the creator, Linus Torvalds, decided to move towards a modular software architecture which would allow him to set boundaries within which the developers of each module have full control over implementation and design details. This architecture countered Brooks' Law and allowed a large group of people to develop a computer program jointly. Torvalds stated that "without modularity, I would have to check every file that changed, which would be a lot, to make sure nothing was changed that would affect anything else. With modularity, when someone sends me patches to do a new file system and I don't necessarily trust the patches per se, I can still trust the fact that if nobody's using this file system, it's not going to impact anything else." [21]. Another study, on the Apache and Mozilla projects, also showed that high modularity (together with many bug finders and fixers (developers)) resulted in low defect densities and

allowed for faster development [23].

### What is a module?

Formally, a module is seen as a self-contained piece of software that contains everything necessary to execute only one aspect of a more broad functionality.

It is worth noting that although there are languages that support modules, the official definition can still differ per language. For example, in Python each file is seen as a module [19], while in Java a module is a collection of packages [18]. Thus, even though all languages use the same general definition of a module, the scale of what is considered a module varies between them.

### Characteristics of modules

Since a module is a self-contained piece of software, it is important that it has as few dependencies to other software as possible. The degree of interdependence between modules is called coupling [24]. In the case of a perfect module, there should be no dependencies between modules. Since there are almost no dependencies to other modules, it is important that the module itself can perform its functionality itself. Making sure that all functionality within the module is tied together and is related is a second requirement of the ideal module. We refer to the degree to which elements inside a module belong together as cohesion [25], which should be high in the case of a perfect module.

Since a module is a piece of software that (ideally) can function without any external requirements, a module is easy to re-use. Using this property, we could state that a single function could also be considered a module, providing it does not require any external dependencies. However, just like the scale of a module used in Java [18], we will not consider this to be a module in this work. We define a module as being a single file with code, just like the official definition used by Python [19].

CHAPTER 3

# Research

Since the main goal of this research is to find differences between modular and non-modular programs, it is important that we are able to define whether a program is modular or not. Thus, one of the main problems to tackle is defining what is considered modularity. After defining modularity, we have to find a way to measure the modularity of programs. And finally, after calculating the modularity of each program, we will perform tests on quality characteristics from the ISO/IEC 25010 standard [13]. After we have calculated both the modularity and quality scores we can plot the results in a graph, using modularity and one of the quality characteristics as the axes. Using the resulting graphs we can analyze the relation between the two characteristics.

## 3.1 Quality characteristics

Even though the ISO/IEC 25010 standard consists of eight characteristics, not all of them are used in this work. We proceed by explaining the rationale for the inclusion/exclusion of these characteristics.

### Compatibility

We do not consider this to be a relevant characteristic for our research since modularity is not likely to have an effect on the way a program could communicate or operate with other systems. Even though it might be easier to add communicative functionality to a modular program, it is not necessarily the case that any API / communicative functionality is available.

### Functional suitability

Functional suitability does not rely on the modularity of a program since the design of the software architecture does not create any functionality that should be available. Therefore we will not analyze the effect of modularity on the functional suitability of a program.

### Maintainability

Since maintainability is an important aspect of software quality, and is (in)directly affected by many properties of the software, we will look at the relation between modularity and maintainability.

### Performance efficiency

Performance efficiency is likely to be affected by the modularity of a program since the execution flow might be different depending on the level of modularity. Since the number of files to load during execution could change depending on the modularity, it is likely that the memory usage, and thus execution speed, will be affected by the modularity of a program. We will analyze whether this is true or not.

#### Portability
Even though portability might be influenced by the modularity of a system. Since the portability of a system is not affected by the software design pattern used, we will not analyze this characteristic. It is, however, the case that a single module within a modular system is easier to migrate to another system since there are (should be) no external dependencies hindering the migration.

#### Reliability
Just like with portability, the software design pattern used does not affect the reliability of a system, since it depends on the code itself, not the design. Thus, we shall not look at the relation between modularity and the reliability of a program.

#### Security
The security of a program is defined by the security measures implemented in the code. Since we are looking at a design pattern rather than code, the relation between modularity and security will not be analyzed in this research.

#### Usability
This characteristic mainly depends on the user experience of a program. As this topic is not part of our research goal, we will not analyze if there is a relation between modularity and usability.

### 3.1.1 Overview

Not all (sub-)characteristics of the ISO/IEC 25010 standard will be analyzed in this research. The most important characteristics for us are maintainability and performance efficiency, whilst the others are either less important or not suitable for this research. Table 3.1 gives an overview of all characteristics.

Table 3.1: ISO 25010 characteristics overview

| **Characteristic** | **Measured in research** |
|---|---|
| Compatibility | No |
| Functional suitability | No |
| Maintainability | Yes |
| Performance efficiency* | Yes |
| Portability | No |
| Reliability | No |
| Security | No |
| Usability | No |

*Except for capacity

## 3.2 Methodology

After defining which quality characteristics will be analyzed, we can look at how we will perform the experiments. We will have to find a set of projects to be tested. For each of these projects we will have to calculate the modularity and the two quality characteristics scores: maintainability and performance efficiency. Using these scores we can perform our analysis on the effect of modularity on the quality of software.

### 3.2.1 Program analysis

In order to analyze each program, we will have to make use of different analyzing methods. If we look at the characteristics that will be tested, performance efficiency requires execution of the program in order to gather information, meanwhile maintainability does not require execution of the program in order to analyze it.

In the first case, we employ dynamic program analysis [26]. Dynamic program analysis is the analysis of software performed by executing the program on a real or virtual processor. In order to get accurate and effective results, dynamic program analysis should be performed multiple times.

In the second case, we employ static program analysis [26]. Static program analysis is the analysis of software performed without executing the program. This form of program analysis is performed on the source code of a program and can give insight in aspects like volume, complexity and modularity (which we will cover below).

### 3.2.2 Modularity

In order to measure the modularity of a program correctly, we have to measure different aspects of a program. We achieve this by using the two main characteristics of a module, low coupling and high cohesion. Since these characteristics tell us something about the dependencies within a program, measuring the dependencies between modules would be an accurate representation of the modularity.

#### Measuring dependencies of a module

Dependencies exist in many forms, such as (imported) variables and constants, but mainly function calls and class definitions. Even though variables or constants used from other modules are also dependencies, we do not consider these relevant in this research. We do not consider them relevant since they do not have a direct impact on the functionality of the program, they mainly provide configuration. In order to measure the modularity of a module, we look at the function calls that exist within the module. We categorize these function calls into internal and external function calls. The former are function calls made to functions within the module itself, while the latter are calls to functions outside of the current module. Once we have gathered all function calls, and categorized them into their respective groups, we can calculate the modularity. We calculate the modularity of a module with the following formula:

$$modularity = \frac{internal\ calls}{all\ calls}$$

This will give us a percentage that represents the modularity of the tested module. After doing this for all modules within a program, we can calculate the modularity for the entire program by taking the average of all separate modules. See paragraph 2.2.1 for more information about modules.

#### Validating the categorization

It is important that the categorization of programs is correct, since any mistake could have a substantial impact on the outcome of the research. In order to validate the categorization mentioned above, we compare the outcomes of the categorizations to that of Better Code Hub [8] and the method using the Modularity Index [10].

### 3.2.3 Maintainability

One of the main code quality characteristics is maintainability. As we have seen, maintainability can be split into the sub-characteristics analysability, modifiability, modularity, re-usability and testability. In order to compute the maintainability score for a program, we first calculate all sub-characteristic scores. In order to calculate these, we use an updated version of existing metrics [11]. The weighted average of scores from other sub-characteristics, which look at code level, are used to calculate the main characteristic scores. Table 3.2 shows the complete metric matrix.

Table 3.2: Maintainability characteristics and calculation weights

| | **Analysability** | **Changeability** | **Modularity** | **Re-usability** | **Testability** |
|---|---|---|---|---|---|
| **Complexity** | - | 4 | - | 3 | 4 |
| **Duplication** | 3 | 3 | - | - | - |
| **Function calls** | - | - | 1 | - | - |
| **Modularity** | - | 2 | - | - | - |
| **Stability** | - | - | - | 2 | - |
| **Test coverage** | 1 | - | - | - | 1 |
| **Unit size** | 2 | - | - | 1 | 2 |
| **Volume** | 5 | - | - | 1 | - |

In order to measure each sub-characteristic defined above, we use an existing model [11] as a guideline. However, we have adapted this slightly to better fit our case. The original metrics that we use for each sub-characteristic can be found in AppendixA.

Once we computed the scores for each maintainability characteristic, we can use these to find the maintainability score for a program using the following formula:

$$Maintainability\ Score = \frac{3 * AS + 2 * CS + MS + 2 * RS + 2 * TC}{10}$$

Where:
$AS$ = Analysability Score
$CS$ = Changeability Score
$MS$ = Modularity Score
$RS$ = Re-usability Score
$TS$ = Testability Score

This formula is based on the metric defined by the Software Improvement Group, as seen in Figure A. Since that model uses the ISO/IEC 9126 definition of maintainability, it is slightly adapted and extended. Our metric uses the weight ratios for analysability and testability as defined in the work by SIG. We extended this by also applying the weight of testability to the changeability and re-usability scores, since they are of the same importance to maintainability. Finally, since modularity is part of maintainability in the ISO/IEC 25010 standard, we included it in the formula. Since we are looking at the relation between modularity and maintainability, the weight of modularity in this formula is kept low since it would show a positive correlation between the two if we did not.

### 3.2.4 Performance efficiency

Performance efficiency consists of three sub-characteristics: capacity, resource utilization and time behavior. For our research we will discard capacity, since the capacity of a program is arbitrary, and only focus on resource utilization and time behavior during normal execution. In order to measure these performance characteristics we make use of dynamic program analysis. We will make a set of projects perform the same functionality for a number of iterations in order to measure the execution speed. After these iterations we are able to take the average time taken for each tool and compare that to the modularity score of each program.

In order to measure the resource utilization we will run each program with the built-in time module of Ubuntu. This will give us some statistics about the execution, we will use the memory usage statistic to plot the resource utilization of each program.

CHAPTER 4

# Experimental setup

Experiments are conducted on projects developed in Python, since Python is a modern and easy to analyze language. Besides providing built-in methods to analyze programs, Python also uses the same definition of a module that we use: a single file. Besides analyzing Python projects, we also perform experiments on Java programs regarding the relation between modularity and maintainability. Just like Python, Java is a popular language, used in many big (open source) projects, making it interesting to analyze. Using the results from all experiments we can see the relation between modularity and the quality characteristics, which can be used to answer the research question. Since some experiments require dynamic program analysis, as mentioned in subsection 3.2.1, we have to find suitable projects that meet our requirements for each experiment.

## 4.1 Limitations

First of all, we only use projects with less than 100k lines. Secondly, we do not use any files that are empty, nor do we use any test files or files place inside test directories. For this exclusion, we assume that test files will either have 'test' in their name, or are situated in a place where a parent folder contains the word 'test' in its name. Besides this we will also ignore files that produce an error while being evaluated. If a big part of all files within a project produce errors, and are thus discarded, we will discard the entire project since we would not be able to give an accurate result for the entire program.

## 4.2 Python experiments

Our Python test suite consists of a total of 39 projects. These projects are all categorized and used in different experiments like method validation and maintainability testing. The full list of projects is available in appendix B. An overview of all experiments is discussed below.

### Method validation

We validate our method of classifying programs using 31 of our 39 Python repositories. By using projects that are somewhat the same in functionality we try to make the validation more accurate and more representative. These projects consist of nine micro web frameworks, seven cryptocurrency related projects and seven IoT projects. The remaining test projects are web crawlers. During these tests we categorized a program as being modular when the modularity score exceeded the used threshold of 0.575 (or 57.5%), which was deemed to be the optimal threshold.

### Modularity vs. Maintainability

After the validation, all 39 Python projects are used to do experiments on the relation between modularity and maintainability.

#### Modularity vs. Performance efficiency

Measuring the differences in performance efficiency between multiple projects with a different modularity score requires a set of projects with the same functionality. Therefore the test suite consists of 11 web crawlers. We will use these web crawlers to extract the first input element from the Google homepage (https://google.com). As mentioned in subsection 3.2.4, we will perform this iteration 100 times in order to get a representative result.

## 4.3 Java experiments

By doing experiments on Java projects we are able to see whether our results from Python are also valid for Java.

#### Modularity vs. Maintainability

Our Java test suite consists of 15 Java projects, which are popular IoT and web crawler projects. Using these projects we analyze whether there are any similarities between the relation between modularity and maintainability in Java and Python. By using IoT and web crawler we use projects that are also analyzed in the Python experiments, keeping some level of consistency between the experiments.

## 4.4 System specifications

Whilst static program analysis is generally not influenced by the operating system or processor, dynamic program analysis does get affected by the system specifications. All experiments have been executed on a laptop with the specifications seen in table 4.1.

Table 4.1: System specifications used in this research

| | |
|---|---|
| **Operating system:** | 64-bit Ubuntu 16.04 LTS |
| **Processor:** | Intel Core i7-4700MQ |
| **Memory:** | 8GiB SODIMM DDR3 (1600MHz) |
| **Cache levels:** | 2565KiB (L1), 1Mib (L2), 6Mib (L3) |

CHAPTER 5

# Results

After performing all experiments, we can analyze the results. Our experiments are categorized into three sections: modularity, maintainability and performance efficiency. In the first section we will use the results from the modularity calculation experiments in order to validate our method of calculating the degree of modularity of Python programs. In the maintainability section we will analyze the results of all sub-characteristics involved in maintainability, and their relation with modularity. The final section will show all results regarding performance efficiency.

## 5.1 Modularity

In order to perform our modularity analysis, we have categorized each program from the test suite into the category modular or non-modular using our own method. This method has been compared to existing methods of categorizing programs as modular or not. The results of these experiments can be seen in table 5.1.

Table 5.1: Method validation results on 31 Python repositories

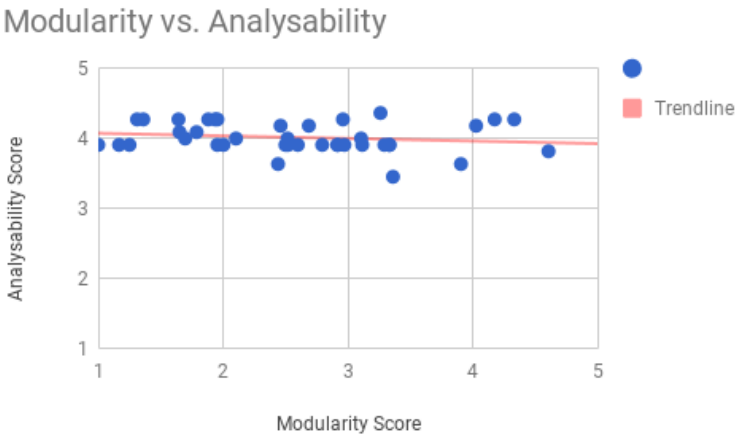
| | **Own vs. MI** | **Own vs. BCH** | **MI vs. BCH** |
|---|---|---|---|
| **False positives** | 4 | 5 | 2 |
| **False negatives** | 4 | 8 | 5 |
| **Corresponding percentage** | 74.2% | 58.1% | 77.4% |

## 5.2 Maintainability

Maintainability consists of five sub-characteristics, which were all calculated for each of the projects in our test set. For each of the sub-characteristics (excluding modularity) a scatter chart was created to visualize their relation to modularity and find a trend line supporting the results. The results for each sub-characteristic are shown below.

### 5.2.1 Analysability

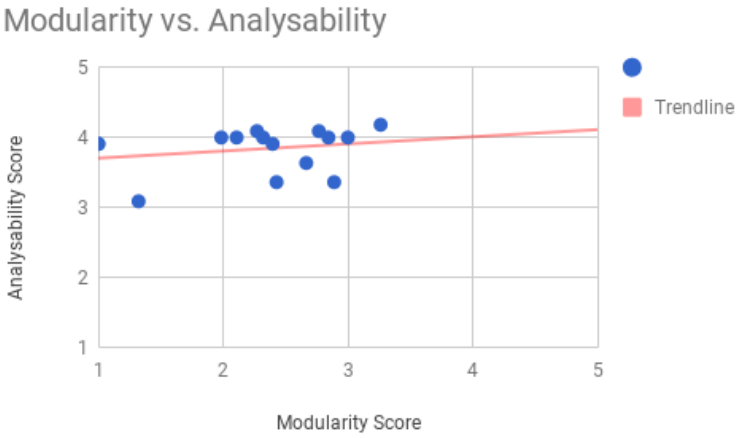
Figure 5.1: Modularity vs. Analysability results in Python




#### Python

As we can see from figure 5.1 there is almost no correlation between the modularity and analysability of a Python program. The Pearson coefficient of -0.163 indicates a slight negative correlation between the two characteristics.

Figure 5.2: Modularity vs. Analysability results in Java

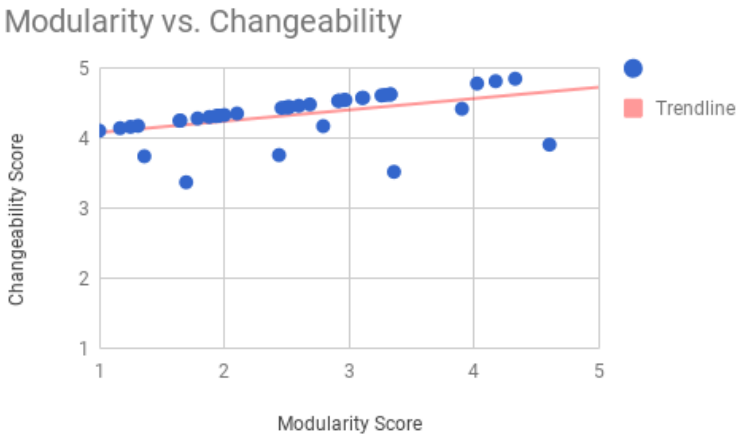



#### Java

Modularity and analysability in Java programs show a slight positive correlation with a Pearson of coefficient of 0.222. The results can be seen in figure 5.2.

### 5.2.2 Changeability

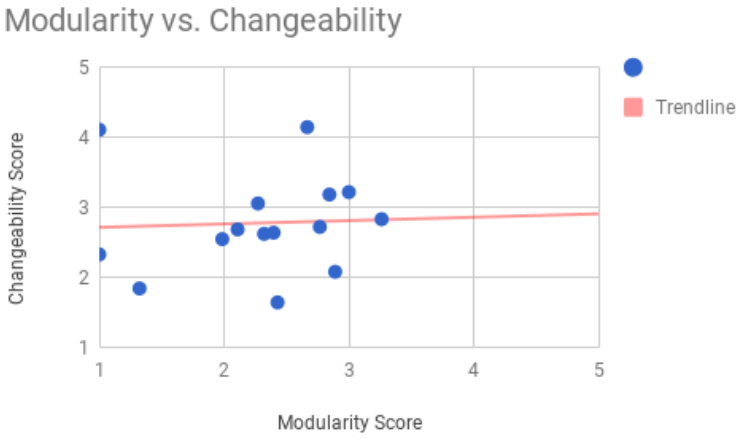
Figure 5.3: Modularity vs. Changeability results in Python




#### Python

The results of this test are shown in Figure 5.3, the correlation has a Pearson coefficient of 0.457, indicating a positive correlation.

Figure 5.4: Modularity vs. Changeability results in Java

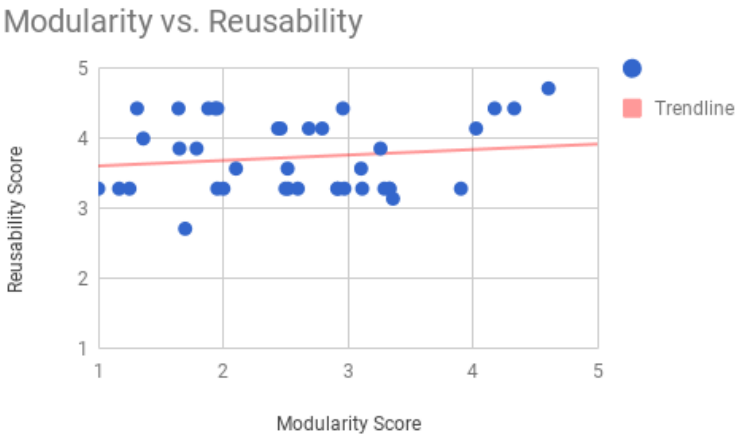



#### Java

With a standard deviation of 0.708 there are a lot of outliers in the relation between modularity and changeability of Java programs. A Pearson coefficient of just 0.048 indicates no relation. The results can be seen in figure 5.4.

### 5.2.3 Re-usability

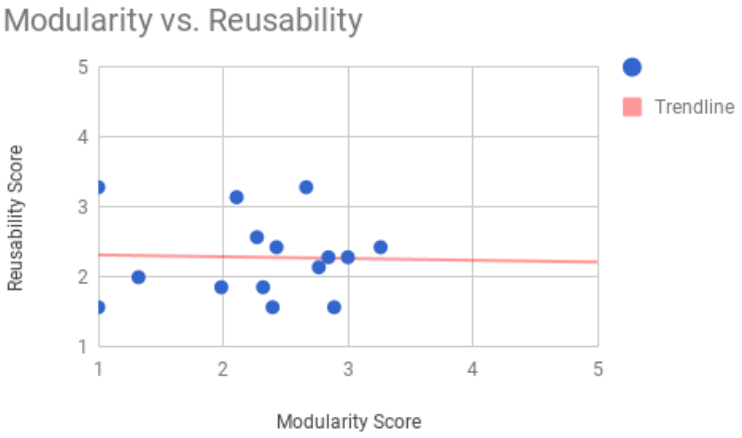
Figure 5.5: Modularity vs. Re-usability results in Python




#### Python

Figure 5.5 shows the results of this experiment. The standard deviation we found is 0.525, which is less compact than the standard deviations of for example the analysability (0.207) and changeability (0.325). The Pearson coefficient we found is 0.137.

Figure 5.6: Modularity vs. Re-usability results in Java

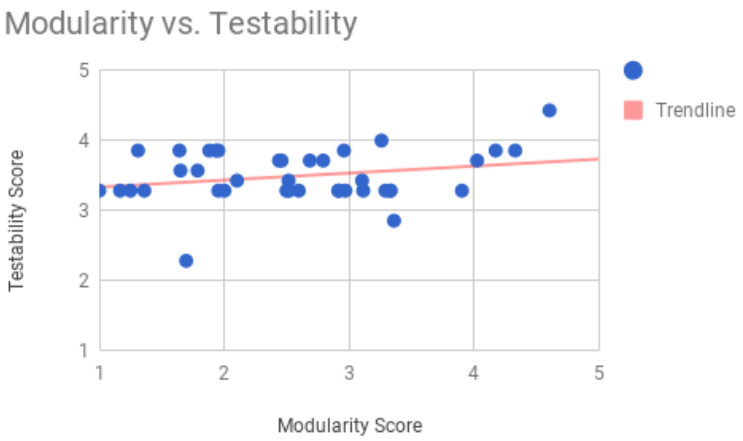



#### Java

Figure 5.6 shows that, just like the changeability, re-usability in Java projects has a large standard deviation (0.589). The Pearson coefficient shows a slight negative correlation between modularity and re-usability: -0.030.

### 5.2.4 Testability

Figure 5.7: Modularity vs. Testability results in Python




#### Python

The relation between modularity and testability in Python projects has a Pearson coefficient of 0.251. Figure 5.7 shows the relation using our test suite.

Figure 5.8: Modularity vs. Testability results in Java


#### Java

Just like changeability, testability in Java shows a large standard deviation of 0.740. The results are shown in figure 5.8. With a Pearson coefficient of -0.183 the relation between modularity and testability is slightly negative.

### 5.2.5 Overall

Figure 5.9: Modularity vs. Maintainability results in Python


#### Python

If we combine these results using a weighted average we get the overall maintainability scores, see figure 5.9.

Figure 5.10: Modularity vs. Maintainability results in Java


#### Java

By combining the results for each sub-characteristic in Java we can get see the overall maintainability relation in figure 5.10.

### 5.2.6 Overview

Table 5.2 contains the results for all tested characteristics involved with maintainability in Python. We can see that most relations show a positive Pearson coefficient, except for analysability. Since the standard deviation is quite low for most relations, the correlation results are quite strong and could be applied to a more broad sense.

All test results from characteristics related to maintainability in Java can be seen in table 5.3. We can see that the relations appear to be less strong by looking at the Pearson coefficients. Meanwhile the standard deviation is quite high in most characteristic, indicating a large difference between projects.

Table 5.2: Overview of maintainability characteristic results in Python

| | Pearson Coefficient | Standard Deviation |
|---|---|---|
| **Analysability** | -0.163 | 0.207 |
| **Changeability** | 0.457 | 0.325 |
| **Re-usability** | 0.137 | 0.525 |
| **Testability** | 0.251 | 0.365 |
| **Maintainability** | 0.521 | 0.277 |

Table 5.3: Overview of maintainability characteristic results in Java

| | Pearson Coefficient | Standard Deviation |
|---|---|---|
| **Analysability** | 0.222 | 0.321 |
| **Changeability** | 0.048 | 0.708 |
| **Re-usability** | -0.030 | 0.589 |
| **Testability** | -0.183 | 0.740 |
| **Maintainability** | 0.163 | 0.416 |

## 5.3 Performance efficiency

The two sub-characteristics of performance efficiency we have tested are resource utilization and time behavior. For both characteristics we calculated the relation to modularity for Python programs and plotted the results in a scatter plot. The results are discussed below.

### Resource utilization

Figure 5.11: Modularity vs. Resource utilization results

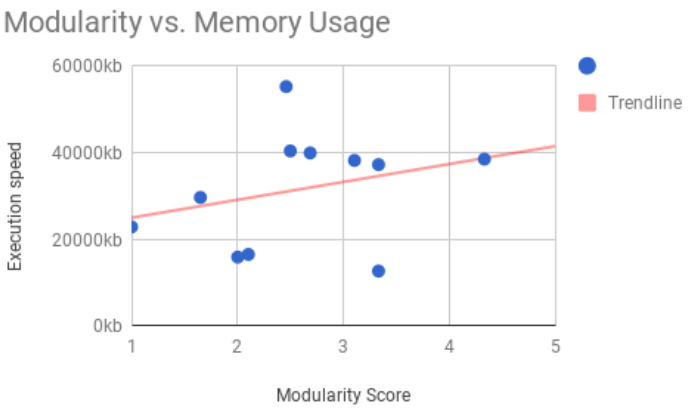
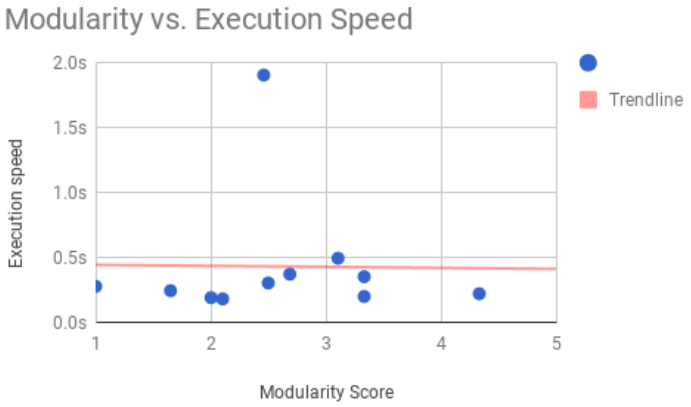



Figure 5.3 shows the result for our experiments regarding resource utilization. The Pearson coefficient of 0.287, shows a positive correlation between the modularity and resource utilization of a program.

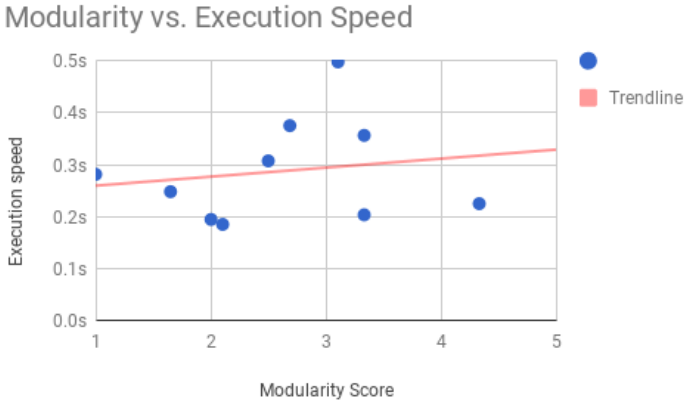
### Time behavior

Figure 5.12: Modularity vs. Time behavior results




The results seen in figure 5.3 show the correlation between modularity and execution speed.

Figure 5.13: Modularity vs. Time behavior results (without outlier)




### Overview

Table 5.4 contains the results for all performance efficiency characteristics that we have tested.

Table 5.4: Overview of performance efficiency characteristics results

| | **Pearson Coefficient** | **Standard Deviation** |
|---|---|---|
| **Resource Utilization** | 0.287 | 13183 |
| **Time Behavior** | 0.169 | 0.099 |

CHAPTER 6

# Discussion

We have measured two main characteristics and their sub-characteristics from the ISO/IEC 25010 standard: maintainability and performance efficiency. By plotting them against the modularity of the programs used we are able to find correlations between them. Besides this, we have validated our own method for calculating the modularity of a program. By comparing the results of our own method to that of existing methods we were able to analyze our own method.

## 6.1 Dependency measuring method validation

As we can see in table 5.1, our method produces the same result as the method using the Modularity Index in almost 75% of all cases. Even though this isn't the ideal value, we can see that this percentage is almost the same for the Modularity Index vs. Better Code Hub approach. We can therefore conclude that even though our method doesn't fully align with neither the Modularity Index nor the Better Code Hub approach, it can be considered correct. One reason why we can say this is because of the flaws we found in both existing approaches. The flaw of the approach taken by Better Code Hub is that the main characteristics of a module are not taken in to account when computing the modularity (score). The Modularity Index, on the other hand, seems to be too complex, making it less reliable. This can be seen when looking at a formally fully modular program (for example a program with just one file in Python). When calculating the modularity using the Modularity Index approach this will produce a result stating that the program is not 100% modular, even though it is. This is partially due to the fact that this method was originally developed for Java systems, which officially defines a module on a bigger scale than Python.

## 6.2 Modularity vs. Maintainability

### Python

The results show that there is in fact a positive correlation between the modularity and maintainability of a program. Whilst not all sub-characteristics of maintainability show a positive correlation with modularity, the overall results show a Pearson coefficient of 0.537, indicating a strong correlation between the two. Thus, we can say that modularity has a positive effect on the maintainability of a program.

The reason why the analysability of a program is negatively influenced by the modularity might be because a more modular program might contain more files, making it harder to analyze all functionality. On the other hand, the changeability of Python programs have a correlation with a Pearson coefficient of 0.457, which indicates a substantial relation (since the Pearson coefficient ranges from -1.0 to 1.0). Thus, we can see that different sub-characteristics might have correlations far apart, but the combination (in this case maintainability) can still give a clear insight.

### Java

When looking at the Java test results for all maintainability-related characteristics in table 5.3 we can see that the correlations appear to be quite weak. With a Pearson coefficient of just 0.163 there appears to be just a slight correlation between modularity and maintainability in Java programs. On the other hand, the large standard deviation in most characteristics involved might indicate that there is no overall relation which could be applied to Java programs. This standard deviation might be caused by a low consistency of expressiveness in Java programs. According to previous research [27], Java has a very inconsistent expressiveness, or efficiency. This fluctuation may have such an effect on the sub-characteristics of maintainability that it makes it very hard to predict a relation between modularity and maintainability for Java systems.

If we look at the sub-characteristics, changeability stands out. With a standard deviation of 0.708, there appears to be no clear relation between modularity and changeability, which is supported by the low Pearson coefficient of just 0.048. The same results appear in the relation with re-usability. A high standard deviation and low Pearson coefficient show us that there appears to be no real correlation.

### The difference between Python and Java

As we can see, Python programs appear to have a positive correlation between modularity and maintainability. With a Pearson coefficient of 0.537 this correlation appears to be quite strong. On the other hand, our experiments have shown that there is just a slight positive correlation between the two characteristics when it comes to Java programs. Another observation is that Python programs appear to follow the pattern more closely. With only one standard deviation of above 0.500 it appears to be much more predictable than Java programs, which have three standard deviations above 0.500. The sub-characteristics appear to be quite different. Unlike Python, modularity and analysability show a slight positive correlation in Java programs, though not very strongly. However, the changeability seems to be positive in both languages. With a Pearson coefficient of 0.457, Python appears to have stronger correlation than Java, which has a Pearson coefficient of just 0.048. Finally, the correlation between modularity and testability differs a lot between Python and Java. Whilst Python shows a Pearson coefficient of 0.251, meaning a positive correlation, Java programs show a negative correlation with a Pearson coefficient of -0.183.

The reason why Java programs show a different result might be because of the way the two programming languages are used. For example, in Java it is required to place public classes in their own files, which in turn results in more files and harder to analyze projects. This is also the reason why our Java test suite shows no highly modular programs and Python does, it is simply required by design to separate functionality into different files, whilst in Python you are free to do what you want. Another aspect that might cause this difference is the expressiveness of both languages, as discussed before. Even though Java has a higher expressiveness than Python [27], Java has a much higher deviation than Python. This difference in efficiency makes it harder to accurately predict test outcomes for certain modularity values, since, as we have seen, the standard deviation of these values is very high.

## 6.3 Modularity vs. Performance Efficiency

When looking at the effect of modularity on the resource utilization and time behavior of a program in Python, we can see that in both cases there is a slight positive correlation between the two. However, the Pearson coefficient values of 0.287 and 0.169 respectively indicate that a more modular program is likely to use more memory than a less modular program, as well as take longer to execute. This might also be due to the fact that a single module might contain more code, since all functionality is grouped together according to the definition of a module. The same could be a cause for the increase in execution time, since it will take longer to execute and load a bigger program.

In the results of the experiments regarding the execution speed, we can see that there is one big outlier which influences the data a lot. With this data point included we get a Pearson coefficient of just -0.015, and a standard deviation of 0.497. This data point uses a different approach to the problem than the others. Whilst the others simply execute our task, this data point first sets up an environment to work with, making it much slower than the others. Since there is a clear difference in execution approach, and thus execution speed, we will remove this outlier. Removing this outlier gives us the results shown in figure 5.3. This figure shows all execution differences on a smaller scale. Using these results we get a Pearson coefficient of 0.169, which indicates a slightly positive correlation between the characteristics.

Overall we can say that modularity has a slight negative effect on the performance efficiency of a program since a positive correlation is present.

CHAPTER 7

# Related Work

Modularity allows for more collaborative software development, and makes it easier to find and resolve bugs [21, 22]. However, we focus whether modularity has any effect on the quality of software, something which cannot be determined from the above. Research has been done on this matter already [28]. In this research, 100 open source C applications were investigated on different quality metrics. As a result, a clear relationship between high modularity and software quality was found. Whilst this follows our hypothesis, there are a few flaws to this research from our point of view. The main flaw is that the metrics used to define the quality of a program using the ISO/IEC 9126 standard [12], whilst we are looking to analyze the programs using the quality definitions from the ISO/IEC 25010 standard [13]. Besides this, the research was focused on open source C applications, while our research mainly focuses on (open source) Python applications. Since the definition of a module is different per programming language, as explained in paragraph 2.2.1.

## Measuring modularity

There are already some methods to measure modularity [8–10], one of these methods is to only look at the coupling between modules [8, 9]. This method looks at the number of incoming calls to a module. This way all external module dependencies need to be evaluated to determine the modularity. However, the flaw to this method is that only incoming calls are evaluated, whilst outgoing calls provide a better insight into the dependencies of a module, since a dependency is a requirement that a module has, not the other way around. It is also the case that the cohesion within a module is neglected, which is important to take into account since it indicates how the different components within a module work together.

### Modularity Index

Research regarding modularity proposed novel metrics in order to specify the modularity of (open source) Java programs [10]. Since it has proven to be effective to use a modular software architecture in open source projects [21], they tried to find a clear metric to justify this. To do this they focused on Java applications, making it somewhat unsuitable for our research. In their metric, an application consists of three layers: classes, packages and systems. Using the modularity that these layers produce, the modularity index can be computed, which defines the modularity of a program. There are however, a few flaws in this approach. First of all, when looking at class level modularity (which in turn influences the other layers) we can see that more metrics are used than just the two core modularity characteristics. This might have an unwanted effect on the outcome of our tests, since a simple program contained in a single module is not seen as fully modular using this approach, even though it follows the module guidelines. This might be due to the fact that in Java a module is by definition of larger scale than in Python: a collection of packages versus a single file. Secondly, the main characteristics of a module (cohesion and coupling) are not seen as the most important factors when computing the modularity of a program. In fact, coupling is not taken into account at all.

### Measuring maintainability

The predecessor of the ISO/IEC 25010 standard is the ISO/IEC 9126 standard [12]. This standard also provided a list of software quality characteristics and their sub-characteristics. Researchers have developed a model for measuring the maintainability of a program using the ISO/IEC 9126 definition of maintainability [11]. They provided a number of metrics that can be used to calculate the sub-characteristics of maintainability, and the maintainability of a program itself. We make use of these metrics by adapting and extending them to fit the ISO/IEC 25010 standard's definition of maintainability. The original metrics can be found in appendix A.

CHAPTER 8

# Conclusion

Writing high quality software is deemed very important in many cases, both in private and corporate projects. It is known that this becomes harder when the size of the project increases [1, 2]. Thus, many big projects make use of a modular software design. One of these examples is the open source project for the Linux kernel [21]. However, does this design technique only improve the development cycle, or does modular programming also have an effect on the quality of the software?

In this thesis we have developed and used a few methods in order to answer this question. We defined quality as the ISO/IEC 25010 standard characteristics [13], of which we looked at maintainability and performance efficiency. By using our own method of calculating the modularity of programs we were able to analyze the relation between modularity and the sub-characteristics involved with these characteristics. Results have shown that there is in fact a positive relation between the modularity and maintainability of a program. Whilst one of the sub-characteristics shows a negative correlation, analysability, the overall maintainability correlation is quite positive with a Pearson coefficient of 0.521 in Python programs. We have tried to see whether this conclusion could also be applied to other languages, such as Java. It seemed however, that this is not the case. With a Pearson coefficient of just 0.163 and a high standard deviation of all data points, this correlation is much weaker and might show us that this cannot be applied to Java programs.

Besides looking at maintainability, the performance efficiency was also measured for Python programs with different modularity scores. The results of those experiments show us that there is also a slight positive correlation between the modularity and performance efficiency of Python software. However, this means that more modular are more likely to use more memory than less modular programs. The execution speed is not directly affected by the modularity of a program. It can, however, be affected by the memory usage, since that would have a (small) effect on the execution speed.

## 8.1 Future work

We have looked at the effect of modularity on maintainability and performance efficiency in Python programs, as well as the effect of modularity on maintainability in Java programs. This research could be expanded by performing experiments on more quality characteristics defined in the ISO/IEC 25010 standard, such as portability and security. Other research that might be interesting is to look at the effect of modularity on quality characteristics in programming languages other than Python. Whilst we have looked at Java as well, the experiments have not been as extensive as in Python. It would be interesting to see whether the same methodology that we used could be applied to more languages to see if a relation could be found there as well. Experiments could be done on languages that work like Python, such as Kotlin and Ruby.

Future research could also include more types of dependencies besides function calls, such as class dependencies and variable dependencies.

# Bibliography


[1] S. Bhatia and J. Malhotra, "A survey on impact of lines of code on software complexity," in Advances in Engineering and Technology Research (ICAETR), 2014 International Conference on. IEEE, 2014, pp. 1–4.

[2] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software complexity and maintenance costs," Communications of the ACM, vol. 36, no. 11, pp. 81–94, 1993.

[3] A. Hunt, D. Thomas, and W. Cunningham, The pragmatic programmer: from journeyman to master. Addison-Wesley Professional, 2000.

[4] T. J. McCabe, "A complexity measure," IEEE Transactions on software Engineering, no. 4, pp. 308–320, 1976.

[5] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of digital computer programs," Communications of the ACM, vol. 6, no. 2, pp. 58–63, 1963.

[6] P. Ralph and Y. Wand, "A proposal for a formal definition of the design concept," in Design requirements engineering: A ten-year perspective. Springer, 2009, pp. 103–136.

[7] T. Barnett and L. Constantine, Modular Programming: Proceedings of a National Symposium. Information & systems Institute, 1968. [Online]. Available: https://books.google.nl/books?id=eI8_HQAACAAJ

[8] S. I. Group. (2018) Better code hub. [Online]. Available: https://bettercodehub.com/

[9] J. Visser, S. Rigal, R. van der Leek, P. van Eck, and G. Wijnholds, Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code. " O'Reilly Media, Inc.", 2016.

[10] A. W. R. Emanuel, R. Wardoyo, J. E. Istiyanto, and K. Mustofa, "Modularity index metrics for java-based open source software projects," arXiv preprint arXiv:1309.5689, 2013.

[11] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," 2016.

[12] "Software engineering – product quality – part 1: Quality model," International Organization for Standardization, Geneva, CH, Standard, 06 2001.

[13] "Systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models," International Organization for Standardization, Geneva, CH, Standard, 03 2011.

[14] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," IEEE Transactions on Software Engineering, vol. 39, no. 8, pp. 1144–1156, 2013.

[15] S. Microsystems, Java Code Conventions, Sun Microsystems.

[16] E. Dijkstra, "On the cruelty of really teaching computing science," 12 1988. [Online]. Available: http://www.cs.utexas.edu/~EWD/transcriptions/EWD10xx/EWD1036.html

[17] Microsoft, Module Class, Microsoft.

[18] P. Deitel, Understanding Java 9 Modules, Oracle.

[19] P. S. Foundation, Python documentation, Python Software Foundation.

[20] F. P. Brooks, "The mythical man-month," 1975.

[21] G. K. Lee and R. E. Cole, "The linux kernel development as a model of open source knowledge creation," unpub. MS, Haas School of Business, UC Berkeley, 2000.

[22] L. Torvalds. (2018) Linux kernel. [Online]. Available: https://github.com/torvalds/linux

[23] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 11, no. 3, pp. 309–346, 2002.

[24] "Systems and software engineering – vocabulary," International Organization for Standardization, Geneva, CH, Standard, 09 2017.

[25] E. Yourdon and L. L. Constantine, Structured design: Fundamentals of a discipline of computer program and systems design. Prentice-Hall, Inc., 1979.

[26] F. Nielson, H. R. Nielson, and C. Hankin, Principles of program analysis. Springer, 2015.

[27] D. Berkholz. (2013) Programming languages ranked by expressiveness. [Online]. Available: http://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness/

[28] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "Code quality analysis in open source software development," Information Systems Journal, vol. 12, no. 1, pp. 43–60, 2002.

# Appendices

APPENDIX A

# Maintainability metrics

Figure A.1: ISO/IEC 9126 code volume metric (source: [11])

| | | KLOC | | |
|---|---|---|---|---|
| rank | MY | Java | Cobol | PL/SQL |
| ++ | $0-8$ | 0-66 | 0-131 | 0-46 |
| + | $8-30$ | 66-246 | 131-491 | 46-173 |
| o | $30-80$ | 246-665 | 491-1,310 | 173-461 |
| - | $80-160$ | 655-1,310 | 1,310-2,621 | 461-922 |
| -- | $>160$ | $>1,310$ | $>2,621$ | $>922$ |

Figure A.2: ISO/IEC 9126 unit size risk level metric (source: [11])

| CC | Risk evaluation |
|---|---|
| 1-10 | simple, without much risk |
| 11-20 | more complex, moderate risk |
| 21-50 | complex, high risk |
| $>50$ | untestable, very high risk |

Figure A.3: ISO/IEC 9126 unit size metric (source: [11])

| | maximum relative LOC | | |
|---|---|---|---|
| rank | moderate | high | very high |
| ++ | 25% | 0% | 0% |
| + | 30% | 5% | 0% |
| o | 40% | 10% | 0% |
| - | 50% | 15% | 5% |
| -- | - | - | - |

Figure A.4: ISO/IEC 9126 code duplication metric (source: [11])

| rank | duplication |
|---|---|
| ++ | 0-3% |
| + | 3-5% |
| o | 5-10% |
| - | 10-20% |
| -- | 20-100% |

Figure A.5: ISO/IEC 9126 test coverage metric (source: [11])

| rank | unit test coverage |
|---|---|
| ++ | 95-100% |
| + | 80-95% |
| o | 60-80% |
| - | 20-60% |
| -- | 0-20% |

Figure A.6: ISO/IEC 9126 maintainability metric (source: [11])

| ISO 9126 maintainability | source code properties: volume (++) | complexity per unit (--) | duplication (-) | unit size (-) | unit testing (o) | |
|---|---|---|---|---|---|---|
| analysability | x | | x | x | x | o |
| changeability | | x | x | | | - |
| stability | | | | | x | o |
| testability | | x | | x | x | - |

APPENDIX B

# Projects used in experiments

Python projects

Flask, pyramid, cherrypy, tornado, aiohttp, klein, falcon, bottle, mmgen, crankycoin, webpy, DarkWallet, bcwallet, pywallet, django-cc, python-trezor, screenly-ose, aws-iot-device-sdk-python, iotedgedev, iot-python, python-gpiozero, goSecure, picamera, cola, demiurge, feedparser, grab, MechanicalSoup, pyspider, robobrowser, scrapy, gain, xcrawler, Zeek, crawlerino, creepy, pholcidae, crawler4py, petterw/crawler.

Java projects

Dropwizard, shazin/iot, agilerules/iot, iot-starterkit, light-rest-4j, mirror, ninja, spark, webmagic, crawler4j, WebCollector, heritrix3, vidageek/crawler, gecco, java-web-crawler.