OPEN UNIVERSITEIT

MASTER THESIS

The numbers about evangelically pursued best practices in PHP projects

Author:
Muhammed Emin AKBULUT
OU student number:
851913244

Supervisor:
Prof. Dr. Marko VAN EEKELEN
Second supervisor:
Dr. Greg ALPÁR

A thesis submitted in fulfilment of the requirements for the degree of Master of Engineering

in the

Open Universiteit

18-04-2019

OPEN UNIVERSITEIT

Abstract

Faculty of Science, Management Technology
Open Universiteit

Master of Engineering

The numbers about evangelically pursued best practices in PHP projects

by Muhammed Emin AKBULUT

The field of software engineering has a great source of scientific research about how to achieve quality using software metrics. There is a great amount of software developers who 'evangelically' pursue quality by proposing best practices with little or no base in academic research. What is the correlation between 'best practices' and the quality in the context of PHP projects? We measured two highly popular best practices, SOLID and Calisthenics, in the top 500 PHP projects in GitHub with 25404 distinct versions. SOLID has a 0.73 correlation score with quality metrics, Calisthenics has a 0.85 correlation score with quality metrics. During our research we created a SOLID rule set to measure violations of the SOLID principles in source code which is a contribution and a unique rule set for the PHP community. We conclude that there is a high correlation between best practices and quality. These best practices do improve the overall quality of software.

Contents

List of Figures

Dedicated to my wife who unconditionally supported me on this beautiful journey

1 Introduction

The field of software engineering has a great source of scientific research about how to achieve quality using software metrics. Quality is used in terms of the maintainability of software. Metrics of software in context of quality are numbers about attributes the source code carries such as the lines of code. Besides the vast amount of researches about quality there is a great amount of software developers who 'evangelically' pursue quality by proposing best practices with little or no base in academic research. These 'opinions' are about what practice could improve quality. Such an opinion can simply be; 'do not use the construct `else` in source code'. The opinions of developers are mostly formed by their experience in software engineering. These opinions and ideas are not always out of the blue, they originate from some book they have read, a conference they have attended in the past or any other source of information or experience.

Software Engineering is an academic research area. So is it possible to research these opinions? Can we validate these opinions? The problem with these opinions is that people who are sceptic or do not like the idea to change their software engineering methods can easily argue that these are just opinions. People who have these opinions mostly want to do things the 'right' way. These sorts of opinions exist almost in all programming language universes. There is research on these topics for Java[23] [15] [13] but PHP on the other side does not seem to be an important topic of research, as we encountered in our literature research.

PHP is a programming language especially popular on the web mainly used for websites. This programming language has evolved from a simple scripting language for a contact form on a website to a complete language with rich options with all kinds of frameworks and libraries. Currently, PHP is used by big companies and small businesses. PHP is feature rich and has a big community. The absence of research interest for PHP can be explained by the area in which PHP and Java are used. Java has found a lot of application within enterprise and government in business-critical software. On the other hand, PHP lacks this widespread usage in enterprise and government environments.

To improve quality within PHP projects continuous research on metrics and other forms of enhancements to maintainability is needed. Other forms of improvement can be automated testing, static code analysis, etc. Quality in PHP is an important topic because PHP grew with the internet and is the most important programming language for companies like Facebook. The question that we want to answer in this research is: What is the correlation between 'best practices' and quality in the context of PHP projects?

2 Research questions

- What is the correlation between 'best practices' and quality in the context of PHP projects? 7.1
  - How can we measure quality? 3.2
    - What metrics can be used to determine quality? 3.2.1
    - What metrics can be used that are applicable for PHP? 4.1
    - What metrics can be used that are feasible quality metrics to use for PHP projects? 4.1
    - What tools are we going to use to measure quality metrics? 3.2.2, 5.1.2
    - How can we capture and store data from quality metrics? 4.3, 5.1.2
  - How can we measure best practices implementations? 4.2
    - What are the best practices for PHP projects? 4.2
    - What are best practices which can be measured in PHP projects? 4.2
    - What tools are available to measure best practices or do they have to be created? 4.2, 5.1.3
    - What tools are we going to use to measure best practices in PHP projects? 4.2, 5.1.3
    - How can we capture and store data from best practices metrics? 4.3, 5.1.3
  - How can we correlate the data measured from the quality metrics and the best practices metrics? 5.3
    - What are the methods to correlate research data in the field of quality? 5.3.1
    - What method can we use to correlate quality metrics with best practices metrics? 5.3.1
    - What results do correlating data provide us? 5.3.2

3 Related work

3.1 Software 'best practices' and evangelism

As mentioned in the introduction there is a lot of evangelism alongside the quality discussion in each software community. Evangelism, as it is called in the community, tells us about the 'best practices' that are told by practitioners. An alternative name we can give these 'best practices' can be myths. These 'best practices' are not scientifically proven or researched but mainly personal or organisational experiences around software development. These best practices can be as simple as not using any form of `else` or `switch` statements in code. Some are more structural in the way a class is structured or the dependencies of a class are organised. We will be researching these 'best practices' in relation to the programming language PHP. There are a lot of 'best practices' ideas. There is SOLID which is a very old and accepted best practice which is first written for C++. Because of the C syntax in PHP the ideas can be used with some modification to it. Another interesting best practice is called Calisthenics. The practice or clue is in its name, it is an exercise to improve quality. The idea behind this is to look further than the available programming syntax and write better code. Again another way of programming to improve quality but not yet proven by academic research.

FIGURE 3.1: validation framework

3.2 Measuring quality

Measuring quality needs a set of quality metrics. After the set is established an environment needs to be established to be able to measure source code with these metrics. There are tools to gather source code metric data. Tools like PHPCS, PHPStan etc. There are also commercial tools available for this purpose. All these steps result in an environment that can be used to determine the validity of programming principles and best practices.

3.2.1 Metrics for quality

There is a great amount of quality metrics research[17][7][6]. These metrics will provide a set for our research. The sub-questions of the research question will need a custom-made metric measuring code. A part of the research will be towards quality metrics.

An international standard is established for quality, better known as ISO/IEC 9126 which revised to ISO/IEC 25010:2011. This standard is about quality but the part we are interested in this standard is the maintainability part. The metric studies we found during our research are pointing to the direction of maintainability. In the following part, we will describe some of the quality studies and especially their views. Their views about how quality could be measured using metrics.

Cavano et al. [8] describe the following set of quality factors.

- Correctness: Does the program satisfies the specifications and fulfills the user's objectives.
- Reliability: Does the program perform in the intended functionality with precision.
- Efficiency: The need for resources by the program to perform its tasks.
- Integrity: In what form is unauthorised use is controlled.
- Usability: The effort needed to understand and use the program.
- Maintainability: The effort needed to locate and fix a bug.
- Testability: The effort needed to test a program.
- Flexibility: The effort needed to modify a program
- Portability: The effort needed to port a program to another platform.
- Interoperability: The effort needed to couple systems to the program.

A program refers to an application in the context above. The paper of Cavano et al. [8] describes a methodology for quality measurements. It proposes a three-layered approach. These three layers are factor, criteria, and metrics. The factors are the above-mentioned categories. The criteria and metrics are related. The metrics form the criteria by being attributes in the qualification of criteria. A factor is calculated by the following formula.

$$r_f = c_1 * m_1 + c_2 * m_2$$

r_f is the rating of the quality factor
c_n is a coefficient

m_n is a software metric.

This calculation method could be used to normalise a range say between 400 and 50 to be a range of 10 to 0. A coefficient could be 0.025.

$$5 = 200 * 0.025$$

A score of 200 will be normalised to 5.

The calculation method mentioned above with a coefficient is widely used by numerous papers, some are [3] [20].

Stamelos et al. use the following set of metrics for quality [22]

1. Number of statements (`N_STMTS`): the executable statement per component
2. Cyclomatic complexity (`VG`): As defined by McCabe [14]
3. Maximum levels (`MAX_LVLS`): the number of levels nested by tabs or spaces.
4. Number of paths (`N_PATHS`): the number of paths in a component.
5. Unconditional jumps (`UNCOND_J`): goto statements.
6. Comment frequency (`COM_R`): the number of comment LOC put out against LOC
7. Vocabulary frequency (`VOC_F`): defined by Halstead [12]
8. Program length (`PR_LGTH`): count the unique operands and operators.
9. Average size (`AVG_S`): average statement size per component.
10. Number of input and outputs (`N_IO`): the points where external I/O is done.

A component is a C function in this context. Each component is tested for these metrics and a method is used to classify each component by testability, simplicity, readability, and self-descriptiveness. These criteria have a relation to the above metrics. As the paper of Cavano et al. [8] describes the metrics are used with a coefficient to determine the score of several groups.

Baggen et al. propose another set of quality metrics [2].

- Volume -> Estimated rebuild value
- Redundancy -> percentage of redundant code (6 lines or more)
- Unit Size -> LOC per unit (unit is an executable piece of code)
- Complexity -> Cyclomatic complexity by McCabe [14]
- Unit interface size -> the number of parameters on each unit.
- Coupling -> The number of calls from the invokers of the methods.

The research uses the ISO/IEC 9126 standard. Baggen et al. [2] uses a star rating system of 1 to 5. The star rating system has the following division (5,30,30,30,5) in percentiles. This division is based on the score a project has in context of the complete set of projects measured. This rating method gives the lowest rating e.g. 1 star for the group of the least 5 per cent and so on. If a score of 100 is used the project

with a score of 4 will be in the first group and will get 1 star and the project with a score of 40 will be classified as a 3-star project. This approach is mainly supported by the notion can be supported by data and is reproducible. This model also suggests quality is defined by existing software. This would bring in the fact that low-quality platform code e.g. in PHP open source networks can create a false sense of quality. The critique about the quality measurement is also mentioned by the authors [2].

In literature review [17][7][6] about the subject of quality metrics, a long list of metrics can be found. Some divide those metrics in categories [17] to be traditional, object-oriented and process metrics. The group traditional is referring to McCabe's [14] and Halstead's [12] metrics. Object-oriented is about the qualities an object-oriented language holds. Process is about the data that could be collected using the process of creating software, such as commit history in version control systems. These researches [17][7][6] conduct a literature review on fault prediction studies, which is part of the quality measurement studies involving metrics. In [17], the McCabe Cyclomatic Complexity is the most successful metric in fault prediction studies. McCabe's Cyclomatic Complexity seems [17] the most successful in object-oriented programming languages. With procedural programming languages, this approach is less successful according to the literature review. Halstead metrics perform not so good [17] there are reports about it being not well. Other metrics that performed well are CBO (coupling between classes), WMC (weighted methods per class) and RFC (Response for a class) [17] object-oriented programming languages [9]. CBO coupling between classes is about the calls in a class to any other method in another class [9]. WMC is the sum of the complexity of all methods in a class [9]. RFC is the sum of external methods called from public methods when initiated [9].

In the literature review of Catal et al. [7] the used metrics in fault prediction studies is mainly based around the method (60%) and the class (24%). Other metric types are file (10%), process (4%), quantitative values (1%) and component (1%). The research also gives out the most used metrics, which are mainly Halstead and McCabe metrics. For our study on quality exact numbers about the quality of software is not our goal. To calculate a coefficient giving us the end result will take a great amount of our research capacity. So the star rating used in [2] is a good method to use for our purpose. The star rating is helpful to determine the overall quality of the tested repositories and this will be our baseline. This star scheme is also adapted by several other researchers to determine quality assessment models [1] [11]. To overcome validation issues for the star rating a big amount of repositories is needed.

3.2.2 Open source tools for quality metrics

The PHP community has a lot of tools available to measure attributes of source code. Some of them are commercial and some of them are open source. These open source tools are very popular and very well supported. This section gives an overview of some of the available open source tools for software metrics, quality, and software code styling.

The common output schemes of all these tools are `XML`, `JSON` or `HTML`. The output of these tools can be easily programmatically combined. The commercial tools for PHP use some of these libraries internally to aggregate data for the user. The challenge will be to create a model with these tools to measure quality in a PHP project.

- `https://github.com/phpstan/phpstan` PHP static analyzer for static analysis (bugs).

- https://github.com/squizlabs/PHP_CodeSniffer PHP code sniffer for style checking. For some research questions possibly new sniffs have to be programmed. This is has a long history of usage in PHP projects.
- https://github.com/sebastianbergmann/phpcpd PHP copy paste detector. It detects the number of copy paste work. This is an attribute used in metrics for quality.
- https://github.com/sebastianbergmann/phploc PHP lines of code. Gives detailed information about the attributes of source code.
- https://github.com/phpmd/phpmd PHP mess detector. It provides metrics of errors and possible bugs in source code.
- https://github.com/Halleck45/DesignPatternDetector A PHP design pattern detector. This tool is currently a proof of concept. It could be helpful in creating detectors for patterns failing to meet the SOLID principles.

These tools all provide a calculation of metrics about quality attributes in the source code of PHP projects.

3.2.3 Commercial tools for quality metrics

- Better code hub: The creators are academic researchers. But the tool does not output any metric data at all. So this tool is not useful to our methodology. The data can only be used in their own methodology.
- Scrutinizer: This tool has no limit on the size of the source code for open source projects. But Scrutinizer has the downside that neither the calculation method nor the metrics are available.

The conclusion can be drawn that both of these tools are not going to be helpful in reaching our research goals. For our research only the open source tools are helpful.

4 Measuring quality and best practices

4.1 Quality metrics

The metrics we need to determine the quality of software in PHP projects lie in the great amount of literature that used plenty of combinations of metrics. The popularity of the metrics in several researches are an indication to us about the effectiveness. Also [17] gives us information about the success of some of the metrics.

The method to choose metrics for this thesis is comprehensiveness meaning that the metric should be calculable with a personal computer. Easy to calculate metrics are those which can be calculated using one source file. Harder to calculate metrics are the ones that need cross-file calculations. Cross-file calculations are highly memory consuming and need a lot of computing power. Another important aspect to choose a metric is the relation to the quality aspect we want to measure. In the following parts we will discuss these issues per metric and the relation to PHP.

The metric LOC (lines of code) is an important indicator of the volume of a project [7] [17] [2] [25] also easily calculable with existing tools. Another important metric is the comment LOC (lines of code), which is an indication of the amount of the documented decisions or explanations [22] [7] in the source code it is also easily calculable with existing tools. The cyclomatic complexity [14] is an indication about the complexity of code [22] [7] [2], complexity requires more effort for a human to understand what is happening in the source code, cyclomatic complexity is easily calculable with existing tools. Duplication in code can possibly create technical debt in multiple locations at the same time [2], this metrics indicates the recurring or duplication of possible bugs it is also easily discoverable in the source code. The unit size in the source code is important because of the grouped command set to achieve a small goal, mainly it is changed and tested as a whole and easily measurable.

The metrics to be used should also be valid to use with PHP. The metric LOC and comment LOC are both relevant in PHP because of the existence of files with multiple lines and also the ability of PHP to incorporate comments in source files. Cyclomatic complexity is also relevant to PHP because PHP is in itself a C syntax based language. It has the structures of `if` `else` `while` etc. Duplication is also relevant for PHP. PHP can be used both in object-oriented way and not. But in PHP functions are essential thus the notion of Unit Size is also relevant to measure. The chosen metrics to determine quality in PHP projects:

- LOC -> lines of code, the lines of executable code.
- Comment LOC -> Comment lines of code, indicated the documentation of a project.
- Cyclomatic complexity -> Measures the paths a function can possibly walk through in execution. [14]

- Duplication -> The redundant lines (6 or more) indicates the technical debt of a project.
- Unit size -> the average size (LOC) per function in PHP, the greater the unit sizes the harder to maintain.

These metrics mentioned above are from several types of researches on quality assessment researches and literature studies [22] [17] [7] and are the most popular in literature of metrics for quality of software predictions. We will use these metrics to determine the classification of a project. The metrics are chosen for the following reasons. The chosen metrics are resource effective to calculate. The metrics are accepted and widely used in the PHP community as well as the academic world. These metrics have multiple implementation and existing software to be determined.

4.2 Best practices metrics

In this section, we will discuss how some of the best practices in the PHP context can be measured and what the restrictions to the measurements will be. We took the SOLID principles and Calisthenics as research subjects. SOLID principles and PHP Calisthenics are the most popular sets of best practices used and preached in the PHP community e.g. in conference talks and user group meetups. There are a lot of resources available of these talks, this gives us more guidance in our search to measure these best practices in PHP. SOLID is also part of classes in university courses. The focus of the measurements are the violations of best practices and the effect on quality metrics of the violated best practices. The choice to measure the violation of best practices is important because the implementation of best practices are hard to detect in software. A detected implementation can also be unintended.

4.2.1 SOLID principles

The SOLID [18] principles are a well-known and popular best practice in the object-oriented programming universe. This makes SOLID a great topic for our research. Despite PHP can be used in a non-object oriented fashion, for the SOLID principles the main focus are the object-oriented projects. This research does not solely focus on object-oriented software. We will be summarising SOLID and its usage for our research. In short, SOLID represents the following principles.

- Single responsibility principle: A class should have only one reason to change.
- The open-closed principle: A class should be expendable without changing methods from the base in the derived class.
- The Liskov substitution principle: A derived class should be substitutable for the base class.
- The interface segregation principle: Make small specific interfaces for only one purpose.
- The dependency inversion principle: Do not depend on concrete classes but on abstractions.

These metrics should be measurable in certain ways because the principles are about certain qualities code holds. Following we will give examples of each principle in a violation and correct implementation setting in PHP derived from the examples given [18] in C++. Issues with the SOLID principles are the alternative implementations to the principles. Not all styles can be measured. But some structures of implementations are common to the PHP universe. We will eventually have false negatives for some of the implementations. A solution will be looking in projects for certain ways of implementing these principles. There is no available measuring rule set for SOLID available for PHP. In our research, we also developed a rule set to measure violations of SOLID principles in PHP. In the following part, there are examples of how to detect violations of SOLID principles in the context of PHP.

Single Responsibility principle
The single responsibility principle can be measured looking at the locations or context of class methods are called. If a class is used by numerous other classes it could have a lot of reasons to change. Also, multiple `public` functions in PHP can be a metric that indicates the violation of the single responsibility principle. The single responsibility principle can be made a metric with this information. As further illustrated in the listings 4.1 the emailService shows two responsibilities by having two public functions which are a clear violation of the principle. The second listing 4.2 shows for the responsibilities of rendering and sending an email two different classes in which they have only one responsibility. These attributes are easily detectable and thus the single responsibility principle can be transformed into a metric.

LISTING 4.1: Single responsibility violation

```
1 class emailService {
2     public function renderEmail($input);
3     public function sendEmail($body, $receiver);
4 }
```

LISTING 4.2: Single responsibility correct implementation

```
1 class emailRenderer {
2     public function render($input);
3 }
4
5 class emailSender {
6     public function send($body, $receiver);
7 }
```

Open Closed principle
The open-closed principle can be measured if extended classes exist. A violation is made when a consumer class of another class depends on the implementation of another class. As seen in the example in listing 4.3. The violation occurs because of the DrawService making the distinction of how to draw a shape with an if statement at line 12 of listing 4.3. The correct way of implementing the open-closed principle is illustrated in listing 4.4. As seen on lines 2 and 6 each figure has its own draw method which makes the DrawService less complex and it has no need to change when a new shape is added. This feature of source code can easily be detected by looking at decision trees of classes using other classes to differ in behaviour.

LISTING 4.3: Open closed violation

```
1 class Circle {
```

```
2       public function getDimensions();
3   }
4
5   class Square {
6       public function getDimensions();
7   }
8
9   class DrawService {
10      public function drawShapes($shapes) {
11          foreach ($shapes as $shape) {
12              if ($shape instanceOf Circle) {
13                  // draw a circle
14              } elseif ($shape instanceOf Square) {
15                  // draw a square
16              }
17          }
18      }
19  }
```

LISTING 4.4: Open Closed correct implementation

```
1   abstract class Shape {
2       public function draw();
3   }
4
5   class Circle extends Shape {
6       public function draw();
7   }
8
9   class Square extends Shape {
10      public function draw();
11  }
12
13  class DrawService {
14      public function drawShapes($shapes) {
15          foreach ($shapes as $shape) {
16              $shape->draw();
17          }
18      }
19  }
```

Liskov Substitution principle

The Liskov substitution is about the ability to use a derived class in context without the code knowing about the derivation. This could be measured looking at derived classes and the usage of their public methods. In PHP this could be measured using the amount off calls using `instanceOf` in contexts of `if` and `switch` as seen on lines 4 and 6 of listing 4.5. In a correct implementation the DrawService should work with all kinds of derivations of shapes without the need of changing the source code as seen as in listing 4.6 line 4. The Liskov substitution principle is measurable by looking at these attributes in PHP.

LISTING 4.5: Liskov substitution violation

```
1   class DrawService {
2       public function drawShapes($shapes) {
3           foreach ($shapes as $shape) {
4               if ($shape instanceOf Circle) {
5                   // draw a circle
6               } elseif ($shape instanceOf Square) {
7                   // draw a square
```

```
8                 }
9             }
10        }
11    }
```

LISTING 4.6: Liskov substitution correct implementation

```
1  class DrawService {
2      public function drawShapes($shapes) {
3          foreach ($shapes as $shape) {
4              $shape->draw();
5          }
6      }
7  }
```

Interface Segregation principle
The interface segregation principle can be measured putting out the amount of interfaces against the number of classes in a project. Also, the function declared in an interface are important attributes which can be a violation of the principle e.g. one or more functions in an interface. As seen in listing 4.7 on line 1,2 and 3 the interface Shape has two different declarations of interfaces for its implementer e.g. Circle on line 6. In listing 4.8 the interface Drawable and Resizeable are examples of interface segregation because they contain only one public function for only one functionality. These attributes are measurable for PHP.

LISTING 4.7: Interface segregation violation

```
1  interface Shape {
2      public function draw();
3      public function resize();
4  }
5
6  class Circle implements Shape {
7      public function draw();
8      public function resize();
9  }
```

LISTING 4.8: Interface segregation correct implementation

```
1  interface Drawable {
2      public function draw();
3  }
4
5  interface Resizeable {
6      public function resize();
7  }
8
9  abstract class Shape implements Drawable, Resizeable {
10     public function draw();
11     public function resize();
12 }
```

Dependency Inversion principle
The dependency inversion principle can be measured by looking at invocations of other classes in a class by looking at the `new` keyword. Another metric could be the use of interfaces in method definitions instead of concrete classes. This can be seen in the context of the `__construct` method which is the point of dependency injection mainly in PHP. An example of a violation can be seen in Listing 4.9 where

the Repository class on line 7 only has the capability of using one type of database as seen on line 1. This can be corrected by depending on interfaces instead of concrete implementations of classes as seen as in listing 4.10 on line 12 where the constructor accepts a ConnectionInterface. The dependency inversion principle is measurable by looking at these attributes in PHP.

LISTING 4.9: Dependency inversion violation

```
1  class Database {
2      public function connect($credentials) {
3          return mysql_connect($credentials);
4      }
5  }
6
7  class Repository {
8      public function __construct(Database $db) {
9          $this->connection = $db->connect();
10     }
11 }
```

LISTING 4.10: Dependency inversion correct implementation

```
1  interface ConnectionInterface {
2      public function connect($credentials);
3  }
4
5  class MySQLConnection implements DatabaseConnection {
6      public function connect($credentials) {
7          return mysql_connect($credentials);
8      }
9  }
10
11 class Repository {
12     public function __construct(ConnectionInterface $db) {
13         $this->connection = $db->connect();
14     }
15 }
```

4.2.2 PHP Calisthenics

Object Calisthenics rules are coding exercises during development of software. Calisthenics should make the code maintainable, the authors claim on several blogs [4][16]. There is also a rule set that can be easily incorporated into our testing and analysing environment. Calisthenics is not a mainstream rule set just like SOLID but the measurements are defined and are stricter and are less prone to produce false negatives. The metrics for Calisthenics are the detected violations of the rules. This is the amount of locations in the source code where violations occur. Calisthenics violations are detectable in all styles of PHP not only in object-oriented PHP code.

A set of Calisthenics rules are illustrated as following by [16]. These are some examples of the overall rule set. Each one is either a violation or a correct implementation.

LISTING 4.11: 1. Only 'X' - X is a variable - defaults to 2 - Level of Indentation per Method - bad implementation

```
1  foreach ($sniffGroups as $sniffGroup) {
2      foreach ($sniffGroup as $sniffKey => $sniffClass) {
3          if (! $sniffClass instanceof Sniff) {
4              throw new InvalidClassTypeException;
5          }
```

```
6       }
7   }
```

LISTING 4.12: 1. Only 'X' - X is a variable - defaults to 2 - Level of Indentation per Method - good implementation

```
1   foreach ($sniffGroups as $sniffGroup) {
2       $this->ensureIsAllInstanceOf($sniffGroup, Sniff::class);
3   }
4
5   // ...
6   private function ensureIsAllInstanceOf(array $objects, string $type)
7   {
8       // ...
9   }
```

LISTING 4.13: 2. Do Not Use "else" Keyword - bad implementation

```
1   if ($status === self::DONE) {
2       $this->finish();
3   } else {
4       $this->advance();
5   }
```

LISTING 4.14: 2. Do Not Use "else" Keyword - good implementation

```
1   if ($status === self::DONE) {
2       $this->finish();
3       return;
4   }
5
6   $this->advance();
```

4.3 The framework

Quality metrics are a greatly sourced academic research area. Best practices are in contrary not much measured or used in academic research. Because of the difficulty of measuring implementation of best practices in code we decided to measure the violations of them. To have a reproducible working method we propose a framework.

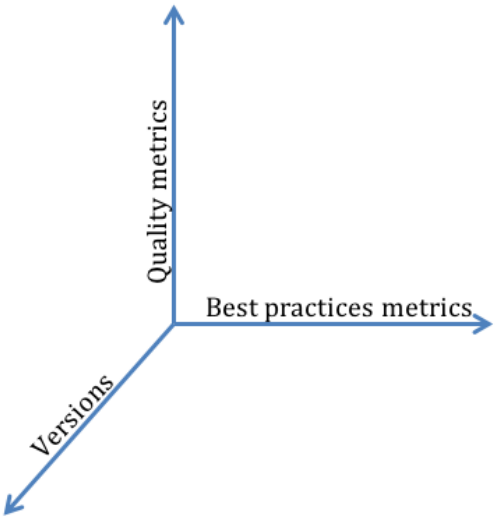
The first objective of this framework is making best practices influence in quality measurable. In our literature research we did not find any framework covering this work field of verifying best practices. This framework will be a contribution to the field of research. Another objective with this framework is creating a reproducible framework so this research should be adaptable to other programming languages and have the opportunity to be free from a fixed set of metrics in our research.

4.3.1 Structure

This framework will describe the collection of two dimensions of data accompanied with the third dimension, which will be the version/tag/snapshot of a certain repository. This third dimension can be established by utilising the availability of historical data in repositories such as GIT, Subversion and Mercurial.

The data of quality metrics, best practices metrics, and the evolution over time with versions of a repository will provide the ability to have a historical view on the metrics.

FIGURE 4.1: framework dimensions



4.3.2 Methods

Quality metrics and best practices metrics are being calculated using the source code of a repository.

Steps involved:

1. Determining the availability of versions/tags/snapshots.
2. Measuring quality metrics for all available versions.
3. Measuring best practice metrics for all available versions.
4. Storing the data
5. Classifying the data
6. Correlating the data

5 The numbers about best practices in PHP projects

In this section, we will discuss the results of our efforts collecting, correlating and analysing data. To collect enough data to analyse we have measured 500 most popular PHP projects from GitHub. These measurements gave us 25404 rows of data from 429 PHP projects.

5.1 Data collection

The data set for the most popular PHP projects is obtained by using the GitHub API [24]. PHP projects are nowadays consumed using Packagist. It is a service for the PHP package manager Composer, created by the creators of Composer. But Packagist as a source could be biased in comparison to the data set that we would obtain from GitHub because of the need for the package manager Composer to use Packagist. It is known that a small portion of highly popular GitHub repositories in the PHP language does not reside on the Packagist universe.

To measure improvements commit history from the GIT repository or version snapshots are available in the majority of the PHP projects. To do this we can use tags of a repository. These tags are used to create version snapshots. If projects do not have version tags available in any way we could ignore historic progress for that particular case. But in popular projects that should not be the case. This historic data is also important because of the improvement in a project, which we want to measure. In the next paragraph, we will cover the implementation of the data set calculation.

To get the tags of a repository we used the following GIT command.

```
1 git tag
```

This GIT command gives a list of all the available tags of a repository. By filtering out all tags that are not in the form of semantic versioning we got a sorted list of tags which can be used to do our further research. These repositories are usable because of the `tags` they contain using a semantic versioning structure e.g. `x.y.z` where `x`,`y`, and `z` are numbers. A sorted semantic versioned list gives a historical development dimension to the collected data.

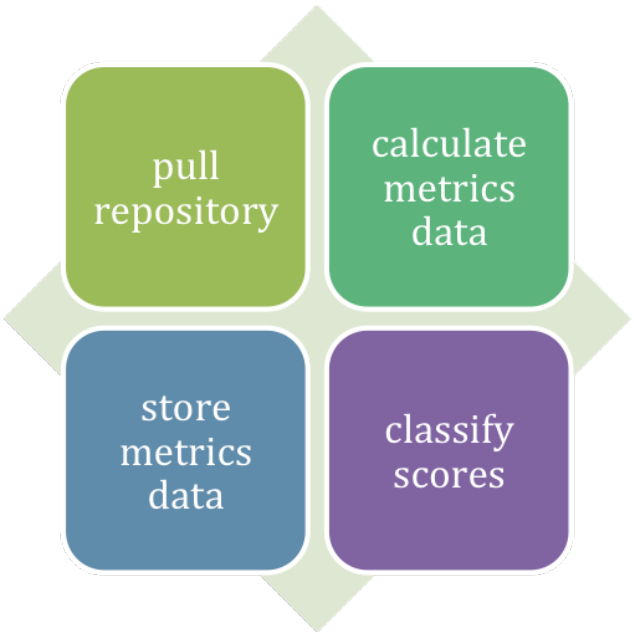
5.1.1 Measuring and analysing environment

To do our research we have implemented the framework into a command line application. Also, we have implemented this in PHP. The choice for PHP is that the tooling for the language is mostly written in PHP and the understanding of the language internals is the strongest among the PHP community we assume. The architecture for the measuring and analysing environment is described in figure 5.1. It will perform the following steps to support our research efforts. Following the framework we described in the previous chapter.

- Clone repository (GitHub repository)
- Checkout tag
- Calculate metrics (quality and best practices) data
- Store metric data
- Classify score

This summary of actions shows the most important parts of the needed environment for our research. The first part is to get the repository, `clone` as in GIT terminology. The tags are listed for a repository. For each tag the metrics of quality and best practices are to be calculated. After the calculation, the data is stored with the meta data from the repository. These metadata are the current measured repository, the current version and if needed other meta information. The data that has been collected must be classified against some quality rules and best practices rules. As we have mentioned before we will use the star rating system to classify each project. After the classification, the correlation of these two sets of data is to be done. The best practices should be easily exchanged or used alongside each other in a modular fashion.

FIGURE 5.1: process



To obtain a GIT repository the following command is used. For our setup, we used SSH authentication with a GitHub account. This makes it possible to not be bound to rate limiting, which could slow down the overall process. The `REPOSITORY_URL` is the location of the repository.

```
1 git clone 'REPOSITORY_URL'
```

To get the code from a certain snippet one must `checkout` to that state of the code in history. The `TAGNAME` is the version we want to measure.

```
1 git checkout 'TAGNAME'
```

Challenges and solutions

In this part we will discuss some challenges we came along during the implementation of the framework and our solutions to them. In previous sections, we discussed some other tools like PHPStan. First, we implemented the least resource extensive tools PHPCS, PHPLOC, PHPCPD. These 3 tools gave us the minimum we needed to do our research. PHPSTAN, however, would be a greater addition to our stack but the downside of this way is that PHPSTAN is resource extensive. It needs the complete application to be installed on the host machine. This would need a lot of downloads of external dependencies into our setup. These dependencies have again system dependencies which we should have met. This would make the effort five times bigger to measure a tag of one repository. For example, the 'symfony/symfony' repository has 78 tags while writing. Using PHPStan can be done for smaller researches but would not be efficient enough for a great field research covering 500 repositories. PHP has no default threading capability, it is single threaded by default. To overcome this challenge and have an efficient way of calculating the metrics we have cut down all the actions to measure in commands to a queuing system. The queued commands are then fed into as many workers as possible. Workers are commands which understand the command in the queue and process the command. This gave us the opportunity to put as much as possible CPU load onto the system. During our trials, we noticed the high CPU loads of our application. We have measured 100 percent load during 24 hours on 2 separate virtual machines which had both 6 vCPU's and 16GB of memory. These specifications for an environment can be found in a normal development desktop or laptop. The memory usage was rather low. Our best effort made one tag of a repository measured in 12 seconds average. The tools we used (PHPCS, PHPLOC, PHPCPD, PHPUnit) to measure are not designed to be used in such a high volume way.

5.1.2 Quality metrics

For quality metrics, we have used 2 distinct tools. PHPLOC and PHPCPD which are both widely used tools to measure quality tools in the PHP community. The following tools are used for the following metrics.

- PHPLOC
  - Lines of code
  - Comment lines of code
  - Cyclomatic complexity
  - Average unit size
- PHPCPD
  - Duplication

PHPLOC and PHPCPD both are easily configurable to run inside an existing PHP application. The following code snippet shows how to get results in code through these tools. The error handling is left out in this example.

```
1 use SebastianBergmann\FinderFacade\FinderFacade;
2 use SebastianBergmann\PHPCPD\Detector\Detector;
3 use SebastianBergmann\PHPCPD\Detector\Strategy\DefaultStrategy;
4 use SebastianBergmann\PHPLOC\Analyser;
5
```

```
6  $finder = new FinderFacade([$repo->getLocalPath()], [], ['*.php']);
7  $files  = $finder->findFiles();
8
9  $analyser = new Analyser();
10 $result = $analyser->countFiles($files, null);
11
12 $strategy = new DefaultStrategy();
13 $detector = new Detector($strategy);
14
15 $clones = $detector->copyPasteDetection($files);
16 $clones->getPercentage();
17
18 $metrics = [
19     'loc' => $result['loc'],
20     'cloc' => $result['cloc'],
21     'cyclomatic_complexity' => $result['ccn'],
22     'duplication' => $clones->getPercentage(),
23     'unit_size' => $result['methodLlocAvg'],
24 ];
```

5.1.3 Best practices metrics

For best practices metrics we used the number of violations occurred in a certain tag of a repository. To detect these violations we have used rule sets for PHPCS. PHPCS can be configured with the usage of `sniffs` by a ruleset.xml which contains all the needed configuration for PHPCS to run on a certain path. In our research efforts, we used the following command to run PHPCS. We ran this on the command line using PHP outside our code because this was the fastest way to get results for PHPCS. PHPCS is not suitable to be incorporated into another application.

```
1 phpcs PATH-TO-SOURCE --standard=STANDARD --report=json --extensions=php
```

In the snippet above `PATH-TO-SOURCE` is the path of the repository to be measured. The `STANDARD` is the path to the directory or the ruleset.xml where the rule set resides, another option is to use a known standard from a list. This list is available in the documentation of PHPCS. The option `report` with the config `json` is to get the results in JSON which gives us the ease to parse the results in a practical way. The option `extensions` is to configure PHPCS to only measure files with the extension '.php' which filters out all unnecessary files such as .txt, .json or any other repository related but non PHP file. For Calisthenics we used an existing ruleset.xml. For SOLID we created a rule set. We did not find a SOLID rule set to be used with PHPCS in our research.

SOLID principles implementation challenges and solutions

In this section, we will discuss the choices we have made during the development of the SOLID rule set. PHPCS runs in a per-file context. It is not possible to combine information throughout the project about e.g. the usage of a class or method. We will discuss the choices and issues we have made during this process. To do this we will use the letters of SOLID as an order. We will start with the S and end with the D.

Single Responsibility principle

Explanation

A class should have only one reason to change. Meaning any conditionals or multi-purpose classes are violating this principle.

Indications

To be looking at what could be an indication in PHP to be a reason for change we came across some pointers.

- Not having an interface which it implements
- Multiple public functions
- Multiple Traits
- Optional parameters in functions

Measuring
Not having an interface or interfaces which the class implements is an indication of lacking a contract to the end-user of the class. This can indicate multiple reasons to change. This will result in a violation of the single responsibility principle. Multiple public functions in a class is an indication of having multiple reasons to change for a class. This will result in a violation of the single responsibility principle. Traits in PHP are a class like constructs which can be used inside classes or functions to add functionality. If a class is using multiple traits this is an indication of having multiple reasons to change for a class. This will result in a violation of the single responsibility principle. In PHP function overloading does not exist such as in C++. Developers tend to use optional parameters in functions to overcome this issue. However, this leads the code to have multiple reasons to change. This is an indication of the violation of the single responsibility principle.

Open Closed principle
Explanation
A class should be expendable without changing methods from the base in the derived class. Without overwriting functionality in derived classes, classes should add functionality without breaking the first intent of the extended classes.
Indication
Open and closed are two but different indications of the principle. The indications we have found for Open Closed principle in PHP are:

- Not implementing interfaces or is not an abstract class (open)
- Not using the keyword final on a class (closed)

Measuring
Open means a class is open for extension of functionality. A class in PHP can be open when it implements an interface or is an abstract class. To detect the violation of the Open part of the principle we look whether it is abstract or implements interfaces. Closed means the source code does not need to change to add functionality. In PHP the `final` keyword is an indication the code cannot be extended by another class to change functionality. If a class is not using the keyword `final` it violates the closed part of the principle.

Liskov substitution principle
Explanation
A derived class should be substitutable for the base class. A new derived class could be created without having to put extra measures in place for other classes depending on the base class. And those other classes should work correctly with the new derived ones.
Indication

In PHP indications for the violation of this rule are to be found in the following items.

- Using of `instanceof`
- Using of `get_class`

Measuring
In PHP these two language constructs are used to determine which class or derivation of a class is dealt with. But this is also an indication of the violation of the Liskov substitution principle because code which is not the actual code of a class depends on that class because it checks whether it a certain class or not to execute some code.

Interface segregation principle
Explanation
An interface has to be single purposed. When a class is created implementing an interface all methods should be implementable for one purpose.
Indication
In PHP there are indications whether an interface is multi-purposed or not.

- Multiple contracts, multiple public functions
- Not implemented contracts (not implemented due to limitations)

Measuring
In PHP an interface can have multiple function declaration, which has to be implemented by the implementing classes. But this is also an indication of the violation of the interface segregation principle. This makes the interface multi-purposed and forces the derived classes to implement it. Another issue with PHP is that classes can be extended and that can make some interface functions implemented in a parent class. It is correct PHP syntax but it can break a class when the parent changes to leave implementing the interface function. So all functions declared in the interface should have been implemented in a derived class. However, this issue is not detectable in the environment we are using to detect violations. The context of the violations can only be the static content of that file.

Dependency inversion principle
Explanation
Classes should not depend on concrete classes but abstractions. Meaning classes should depend on the contracts created with abstractions.
Indication
These abstractions in PHP are in the form of interfaces or abstract classes. In PHP we can detect the following attributes.

- Dependencies in a function call are concrete classes
- Object instantiations in classes

Measuring
If dependencies in a function call are concrete classes the dependency inversion principle is violated. In the tool, we are using and the way PHP projects nowadays are set up we cannot easily discover whether a class is abstract or an interface. To be able to have some sort of violation detection we are depending on the implementers' good practice of calling the class or interface 'abstract' or 'interface' in the name. Object

instantiations in classes in PHP depend on concrete implementations and are called with the keyword ‘new‘ like ‘new stdObject();‘ this gives a direct dependency to a class which is a violation of the dependency inversion principle.

5.1.4 Results

We selected the most popular projects for PHP in GitHub using their star ratings [10]. 500 repositories from this list were fed into our application. This gave us 429 usable repositories. These repositories contained valid semantic versioned tags. From these repositories, we got 25404 rows of data. This gives us 177828 data points for our further research. Which is a significant amount for our further research.

5.2 Data classification

For data classification, we use the method of Baggen et al. [2]. The star rating needs some further calculation on the raw data we collected.

5.2.1 Methods

To classify the scores of each metric, percentile calculations and correction on outliers is needed.

To calculate percentiles we first remove the outliers[19]. After the cleanup we calculate the percentiles according the star rating. Because some metrics are higher the better or lower the better, the percentile order of the numbers can differ. Also another important feature we discovered with the duplication metric is that a great amount of repositories have 0% duplication in them. Additionally to the outliers the 0 values are also filtered because it rendered the star rating useless.

5.2.2 Results

We classified all the metrics we collected to the star rating model. These are our ratings created with the data we have collected from all the repositories we have measured. 1 is the lowest and 5 the highest star rating. After this classification all the data we collected is rated using these threshold values.

star	threshold values
1	>= 157635
2	< 157635 and >= 19950
3	< 19950 and >= 5336,5
4	< 5336,5 and >= 744
5	< 744

TABLE 5.1: LOC threshold values

star	threshold values
1	>= 39530,5
2	< 39530,5 and >= 5043
3	< 5043 and >= 1241
4	< 1241 and >= 129
5	< 129

TABLE 5.2: CLOC threshold values

star	threshold values
1	>= 8233,2
2	< 8233,2 and >= 858
3	< 858 and >= 229
4	< 229 and >= 24
5	< 24

TABLE 5.3: Cyclomatic complexity threshold values

star	threshold values
1	>= 6.19
2	< 6.19 and >= 4.38
3	< 4.38 and >= 3.46
4	< 3.46 and >= 2.46
5	< 2.46

TABLE 5.4: Average unit size threshold values

star	threshold values
1	>= 4.27
2	< 4.27 and >= 1.28
3	< 1.28 and >= 0.57
4	< 0.57 and >= 0.14
5	< 0.14

TABLE 5.5: Duplication threshold values

star	threshold values
1	>= 2573
2	< 2573 and >= 494
3	< 494 and >= 140
4	< 140 and >= 10
5	< 10

TABLE 5.6: SOLID threshold values

star	threshold values
1	>= 11210
2	< 11210 and >= 1046
3	< 1046 and >= 254
4	< 254 and >= 17
5	< 17

TABLE 5.7: Calisthenics threshold values

5.3 Data correlation

5.3.1 Methods

The method we use for data correlation is the Spearmann[21] method. Spearmanns method is designed to correlate ranked data. Correlation output ranges from -1 to 1 and values lower then -0.6 and higher then 0.6 are valued as being a strong correlation. The formula for Spearmann method we used is as following.

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)} \tag{5.1}$$

5.3.2 Truth behind the evangelism

For the correlation, we used the quality metrics in their rated form. We have correlated each metric to the best practices metrics and a combined rating of the quality metrics against the best practices metrics. The results of the correlation of these rated data can be found in the following table.

metric	metric	correlation result
LOC	Calisthenics	0.88
CLOC	Calisthenics	0.85
Cyclomatic complexity	Calisthenics	0.89
Duplication	Calisthenics	0.52
Average Unit size	Calisthenics	0.38
Combined scores	Calisthenics	0.85
LOC	SOLID	0.85
CLOC	SOLID	0.84
Cyclomatic complexity	SOLID	0.78
Duplication	SOLID	0.48
Average Unit size	SOLID	0.18
Combined scores	SOLID	0.73

TABLE 5.8: Correlation for quality and best practices metrics

As it can be read from table 5.8 there is a strong correlation between the combined scores and the best practices Calisthenics and SOLID. These results show also the difference between Calisthenics and SOLID. Calisthenics is an effort to reduce some of the metrics we used for our research. Calisthenics has the effort to reduce cyclomatic complexity and the average unit size. The results are showing us that Calisthenics has stronger correlation on those two metrics then SOLID has. SOLID is a more architectural best practice set where Calisthenics is focused on readability and thus maintainability where our maintainability metrics perform well on.

The combined score is calculated using the rounded average of the five metrics we used to measure quality. We rounded this value because we wanted to adhere to the rating we established for the metrics. This is also required for the correlation technique. Therefore the correlation value is not the average of the correlation values of the five metrics against the best practices metrics.

The metrics Duplication and Average Unit Size do not have a high correlation value as much as the other metrics with both SOLID and Calisthenics. Both SOLID and Calisthenics are not focusing on creating less duplication inside a repository is an explanation for this weak correlation. But Calisthenics is focusing on lesser

Average Unit Size. The difference of 0.20 of Average Unit Size correlation where Calisthenics has a higher correlation then SOLID can be explained by this. We can also conclude that most of the projects are not focusing as much as on lesser Average Unit Sizes then, for example, try to have less Cyclomatic Complexity.

The numbers behind the best practices show that the evangelism is not misplaced. SOLID and Calisthenics do improve the quality of PHP projects.

6 Validation of results by other metrics

In this part, we will discuss our validation efforts for the results we found in the previous chapter. The validation we seek is that other data also can correlate successfully on the collected data.

6.1 Validation metrics and data

To validate the results from the previous correlations we are using other metrics used in quality research. Contributors on repositories represent the amount of persons contributed to the code. Another metric are the forks of the repositories. Forks represent contribution efforts and are also a control metric. Test coverage of a repository is also collected.

This data is collected during our data collection efforts of quality metrics and best practices metrics. The data of the forks is acquired from GitHub where this information is listed in the list we obtained to determine the most popular 500 repositories.

The data for the contributors is collected using GIT. The following commands are used to collect the number of contributors in a git repository.

```
1 git log --all --format=\'%aN\' | sort -u | wc -l
```

For our third metric, the test coverage is also interesting because of the complexity of this metric to measure. All the previous metrics in this thesis are all calculable in all contexts in PHP. But the test coverage highly depends on the software stack is used by the developers of the repository we measure. Just like other issues we mentioned before with PHPStan the results can be strongly biased because of the usage of tools like the package manager Composer. To measure all the tags and all the repositories is from our experience a highly challenging job for this thesis and out of scope. To have data to correlate we investigated what could be the least amount to correlate to. A number of 325 tags and 75 repositories is a good amount of points to correlate[5].

The steps involved to measure a test coverage cover the following steps. These steps are repeated for every tag of every repository we measure.

- clone git repository
- checkout to a tag via git
- install all dependencies via Composer
- run PHPUnit
- collect the coverage percentage

The complexity aspect for measuring test coverage is installing all dependencies. To be able to do this the following requirements have to be met. Commonly most of

the repositories have different requirements and sometimes these requirements are in conflict with the system we are using.

- PHP version
- PHP libraries
- repositories

Our setup requires Composer and PHPUnit to measure. We used PHP 7.2 and Xdebug 2.6.1 to measure the test coverage of a library using PHPUnit. Older versions of repositories can have the constraint that it does not work with PHP version 7. The biggest issue when measuring the test coverage was that the requirements of older tags faced the problem that not all the requirements could be met anymore because of the newer PHP versions, abandoned repositories or even badly configured Composer files. Some repositories had also a lot of interesting additional configurations in them to meet their requirements to run the test successfully in their release strategy. After a lot of trial and error, we created a stable environment to measure the test coverage. Where on average 12 seconds are spent on measuring all other metric in this thesis, test coverage took around ten times more time to measure. Because of this knowledge on our first small proof of concept we decided to measure fewer repositories and fewer tags. We eventually measured 248 repositories where we ended up with usable results of 75 repositories. Because of the great amount of time it takes to measure a repository tag we decided to not measure all bugfix tags of a repository. Only the minor tags are used for measurement. Major, minor and bugfix are the commonly used semantic versioning tag numbers.

Measuring the test coverage we also stored every output into a text file.

```
1 composer install --no-interaction --quiet --no-scripts --no-progress --no-
    suggest
2 phpunit --coverage-text
```

Issues with Composer where the great amount of pre and post scripts during installation, these scripts are for numerous reasons such as cache warming. Also, output of progress or suggestions can cause our setup to fail. The ability Composer gives a developer to change the default install directory of dependency in particular for PHPUnit was another big challenge to locate the PHPUnit executable file. Also some older versions of PHPUnit where installed along the way. An older PHPUnit could be missing the required reporting options.

The biggest threat to the validity of the test coverage metric is that its only available for some of the projects and not all projects. Where other metrics can be calculated for all the projects.

6.2 Validation results

The validation metrics we have collected are also ranked. Here are the ranking results.

star	threshold values
1	<= 31
2	> 31 and <= 223
3	> 223 and <= 564
4	> 564 and <= 1682
5	> 1682

TABLE 6.1: Forks ranking

star	threshold values
1	<= 19
2	> 19 and <= 88
3	> 88 and <= 145
4	> 145 and <= 406
5	> 406

TABLE 6.2: Contributors ranking

star	threshold values
1	<= 62.99
2	> 62.99 and <= 84.46
3	> 84.46 and <= 94.54
4	> 94.54 and <= 99.94
5	> 99.94

TABLE 6.3: Test coverage ranking

The results from the correlation between the combined scores and the validation metrics show very low correlation. Thus we can not conclude that there is a correlation between these validation metrics and the combined score metric of the quality metrics. However, we see a negative correlation between the forks and contributors against the combined scores. This means that the forks and contributors move the opposite direction in terms of development. This would tell us the quality metrics get better when forks and contributors are lesser.

Other results we can derive from these numbers is that the myth in the software development world that a high test coverage leads to higher quality software is also proven wrong.

metric	metric	correlation result
Combined scores	Forks	-0.29
Combined scores	Contributors	-0.38
Combined scores	Test coverage	0.11

These results are not invalidating the correlation results we have shown in previous chapters for best practices metrics and quality metrics. However, validating those result need more effort but that is out of scope for this thesis.

7 Conclusion and further work

7.1 Conclusion

The great resource on quality literature has given us great handles on how to measure quality in software. Alongside these metrics, the interesting field of best practices has less literature coverage. Especially in the PHP community there are many best practice evangelists. But, is there a correlation between quality and best practices in the context of PHP projects? Best practices are preached by practitioners rather than scientist in the field of software engineering. These best practices have been proven by these practitioners in their own work field or they believe it would benefit the quality. SOLID and Calisthenics are two most popular sets of best practices and especially these two sets are commonly discussed in conferences and user group meetups.

There are numerous quality metrics designed over the years. In this research, we used the five most popular metrics. The metrics we used are lines of code, comment lines of code, cyclomatic complexity, duplication and unit size. For the best practices, we measured the violations of the rules set derived from best practices. Detecting the implementation of best practices is far more complex than detecting violations of best practices. To have reproducible research we looked for frameworks which can facilitate our research efforts. However, we could not find one that would fit this research. Our contribution to this field of research is a framework which helps to systematically research the field of best practices. This framework gives the ability to not only measure both the quality and best practices metrics but also historic data on these numbers by using data from version control systems such as git. Another contribution we have made is the creation of a SOLID rule set which can be used in the daily work of a PHP developer to detect SOLID violations in combination with PHPCS. The best practice rule set Calisthenics has already some rule sets available on GitHub.

With the framework we have measured metrics for 500 projects, which are popular on GitHub. From these 500 project 429 projects resulted in usable data. This again resulted in 25404 unique rows of data. Each row contains 11 unique values of metrics. For the correlation of such diverse data we rated every value by calculating a star based on a star rating system which is a percentile classification of a value. These star ratings are then correlated using the Spearmann method. This resulted in interesting findings.

To validate our findings of best practices we used some metrics and tested their correlation just like the best practices metrics on the quality metrics. These metrics are the amount of forks, the amount of contributors and the test coverage percentage of a repository. The forks and contributors can easily be collected. But the test coverage metric was the hardest metric to measure in this research. This was also because of the dependency of the great number of different configurations in projects. In this validation effort we did not find any data that could validate our previous results. But we found out that e.g. a high test coverage is not a sign of high quality. Which is also a myth in the software community.

We have found a correlation of 0.85 of the combined quality scores and Calisthenics, this number indicates a high correlation. Between the combined quality scores and SOLID the correlation of 0.73 shows a high correlation. From the high numbers of correlation between quality metrics and best practices metrics we can conclude that there is a strong correlation between these metrics we used for our research. This correlation gave us also some other interesting results. Calishtenics has a very high correlation on cyclomatic complexity and unit size quality metrics. Calisthenics therefore really proves its point, Calisthenics is mainly targeting complexity through cycles in code and unit size. Also SOLID as a best practice can help to improve quality. Architectural focus can help software to be less cluttered resulting in fewer lines of code, less cyclomatic complexity etc. best practices do improve the overall quality of software. The SOLID rule set we created during this research is also an unique and an important contribution to the PHP community. This rule set is also being validated in this research by the quality metrics.

7.2 Discussion

What is the correlation between 'best practices' and quality in the context of PHP projects? Is the question we asked starting our research. We found correlations between best practices and quality. We also could not find a good validation metric for our correlation findings. For the best practices metrics we used the amount of violations of the best practices in the source code. Are violations the right way to look for a relation between quality and best practices? One can argue that the number of violations tend to increase over the increasing lines of code. Also, the lines of code quality metric correlated well on the best practices metrics, underlining this argument. The best practices metrics correlated well on quality metrics which are the metrics indicating the volume. These metrics are lines of code, comment lines of code, and cyclomatic complexity. However, cyclomatic complexity tends to increase over the number of lines of code. Cyclomatic complexity still is a significant metric which can point out code which is hard to maintain. The amount of code, e.g. lines of code, actually makes it harder to have a grasp on the project for a developer. This feature makes maintaining code harder. The tooling we used to measure best practices metrics is another point of interest. In this research we used PHPCS over PHPStan. PHPStan can give much better and far more detailed metrics about best practices metrics. Our choice for quantitative research, 500 projects, led us to use PHPCS which performs faster and using less resources. The effect of this choice on our results is currently unknown. We believe the adaptation of PHPStan to a research like this can give resourceful insights. Another much heard best practice in the PHP community is to keep the unit size low. Also, the unit size has a weak correlation on the best practices metrics. These results are showing us that best practices definitely do not always help developers to improve the maintainability of the code they write at all fronts. We used two sets of best practices in our research but maintainability is hard to cover in just one set. Or alternatively stated, the focus of a best practice does not cover all the areas of maintainability metrics. This contradicts the claim that best practices improve the overall code quality. Best practices can also advice the opposite of each other. There is definitely no silver bullet to improve maintainability. The results of our validation efforts show very weak correlation on quality metrics. Finding good validation metrics is another big area to cover which fell outside the scope of this research. These validation results can tell us something new. The number of forks and contributors are not likely to show the quality of a project.

In the open source community people evangelise that many forks or contributors shows the detailed interest in the quality of a project. We can tell the popularity of a project by forks and contributors but this does improve the quality. We should not forget the people who decide someone can contribute to a project have also a great effect on quality when accepting these contributions. The maintainers of the project should also be aware of and the gate keepers of the quality of the contributions. In essence this research proves the value of best practices in creating software. Best practices can add quality to software and this research proves that point. The mythical proportions these best practices are getting by their promotion by evangelists in the PHP community can create scepticism among some developers. But nevertheless, there is actual value to be practising these best practices.

7.3 Further work

The way we measured best practices metrics is the way with least resistance. We did not execute the code but only did static analysis. For the test coverage, we only looked into unit tests. Collecting data about tests has also the pitfall to only cover projects that are designed to be testable. A project missing the test configuration or even tests can be a truly good project by adhering only to static code analysis. Also, static code analysis tools in PHP are now rising stars. Especially PHPStan is developing at high speed. In the days this research has begun it was new but now it is fully grown and adapted by many major vendors in the open source community. This tool can now give insights into bugs in PHP projects that are not always discovered using tests. PHPStan can also incorporate best practices rule sets. This can give the possibility to detect SOLID violations more accurately then our efforts using PHPCS. Also with the development of the newer versions, PHP is getting more and more statically typed, it is optional however. Other interesting steps for quality research is to improve code by using best practices in an automated fashion. This could result in programming best practices where software can detect for example unnecessary cyclomatic complexity and reduce it. There are tools capable of refactoring code where this can be integrated into with also showing the improvement on quality metrics. The greatest challenge is to use the best practices in improving the code and calculate which best practice can give the most improvements to a project in real time. Besides best practices e.g. anti-patterns in software could also be detected and repaired in real-time. Also, real time calculation on quality improvements can be shown to the developer. Software proposing improvements in real time backed by metric data instead of templates can be ground breaking. For other programming languages this research can also be applied using the framework we have created. There are a lot of programming languages with a great open source community. Where also a great source of open source and commercial tools exists to measure quality. GitHub is a great source of open source projects for a research.

A Appendix: Software and Data

All software used for this thesis and data can be found in two github repositories. Enclosed with a README to reproduce the outcome of this thesis.

Thesis app contains the software used for this thesis.
https://github.com/muhammedeminakbulut/thesis-app

Thesis data contains the data used for this thesis.
https://github.com/muhammedeminakbulut/thesis-data

Articles

[1] Jehad Al Dallal. "Object-oriented class maintainability prediction using internal quality attributes". In: *Information and Software Technology* 55.11 (2013), pp. 2028–2048.

[2] Robert Baggen et al. "Standardized code quality benchmarking for improving software maintainability". In: *Software Quality Journal* 20.2 (2012), pp. 287–307.

[3] Victor R Basili, Richard W Selby, and David H Hutchens. "Experimentation in software engineering". In: *IEEE Transactions on software engineering* 7 (1986), pp. 733–743.

[5] Mohamad Adam Bujang and Nurakmal Baharum. "Sample Size Guideline for Correlation Analysis". In: *World Journal of Social Science Research* 3 (Mar. 2016), p. 37. DOI: 10.22158/wjssr.v3n1p37.

[6] Cagatay Catal. "Software fault prediction: A literature review and current trends". In: *Expert systems with applications* 38.4 (2011), pp. 4626–4636.

[7] Cagatay Catal and Banu Diri. "A systematic review of software fault prediction studies". In: *Expert systems with applications* 36.4 (2009), pp. 7346–7354.

[9] 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.

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

[17] Danijel Radjenović et al. "Software fault prediction metrics: A systematic literature review". In: *Information and Software Technology* 55.8 (2013), pp. 1397–1418.

[20] Richard W Selby, Victor R. Basili, and F Terry Baker. "Cleanroom software development: An empirical evaluation". In: *IEEE Transactions on Software Engineering* 9 (1987), pp. 1027–1037.

[21] Charles Spearman. "The proof and measurement of association between two things". In: *American journal of Psychology* 15.1 (1904), pp. 72–101.

[22] Ioannis Stamelos et al. "Code quality analysis in open source software development". In: *Information Systems Journal* 12.1 (2002), pp. 43–60.

Conference proceedings

[8] Joseph P Cavano and James A McCall. "A framework for the measurement of software quality". In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 7. 3-4. ACM. 1978, pp. 133–139.

[10] Laura Dabbish et al. “Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository”. In: *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*. CSCW ’12. Seattle, Washington, USA: ACM, 2012, pp. 1277–1286. ISBN: 978-1-4503-1086-4. DOI: 10.1145/2145204.2145396. URL: http://doi.acm.org/10.1145/2145204.2145396.

[11] Francesca Arcelli Fontana et al. “Automatic metric thresholds derivation for code smell detection”. In: *Emerging Trends in Software Metrics (WETSoM), 2015 IEEE/ACM 6th International Workshop on*. IEEE. 2015, pp. 44–53.

[13] Radu Marinescu. “Detection strategies: Metrics-based rules for detecting design flaws”. In: *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE. 2004, pp. 350–359.

[15] Matthew James Munro. “Product metrics for automatic identification of "bad smell" design problems in java source-code”. In: *11th IEEE International Software Metrics Symposium (METRICS’05)*. IEEE. 2005, pp. 15–15.

[23] Eva Van Emden and Leon Moonen. “Java quality assurance by detecting code smells”. In: *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE. 2002, pp. 97–106.

[24] Simon Weber and Jiebo Luo. “What makes an open source code popular on git hub?” In: *Data Mining Workshop (ICDMW), 2014 IEEE International Conference on*. IEEE. 2014, pp. 851–855.

[25] Hongyu Zhang. “An investigation of the relationships between lines of code and defects”. In: *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE. 2009, pp. 274–283.

Books

[4] Jeff Bay. *The ThoughtWorks Anthology: Essays on Software Technology and Innovation, Chapter Chapter 6: Object calisthenics*. O’Reilly Series. Dallas, TX: Pragmatic Bookshelf, 2008.

[12] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977. ISBN: 0444002057.

[19] Peter J Rousseeuw and Annick M Leroy. *Robust regression and outlier detection*. Vol. 589. John wiley & sons, 2005.

Others

[16] *PHPCS calisthenics rules*. https://github.com/object-calisthenics/phpcs-calisthenics-rules/.

[18] Robert C. Martin. *Principles of OOD*. http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOo (May 2005).