Design Debt Prioritization

A Design Best Practice-Based Approach

Reinhold Plösch
Johannes Kepler University Linz
Linz, Austria
reinhold.ploesch@jku.at

Johannes Bräuer
Johannes Kepler University Linz
Linz, Austria
johannes.braeuer@jku.at

Matthias Saft
Corporate Technology Siemens AG
Munich, Germany
matthias.saft@siemens.com

Christian Körner
Corporate Technology Siemens AG
Munich, Germany
christian.koerner@siemens.com

ABSTRACT

Technical debt (TD) in a software system is a metaphor that tries to illustrate the remediation effort of the already introduced quality deficit and the impact thereof to the business value of the system. To address TD, various management activities are proposed, each addressing a particular purpose. Whereas the activities of debt identification and measurement are broadly considered in literature, the activities of debt prioritization and communication lack appropriate approaches with an economic perspective. This work proposes a TD prioritization approach. Therefore, it narrows down the focus of TD to design debt and relies on the quantification of design best practices. Further, the non-conformance of these practices is assessed by applying a benchmarking technique. As a result, the gained information is transferred into a portfolio-matrix to support the prioritization and communication of design remediation actions. The applicability and suitability of the approach are demonstrated by using the source code of the open source project GeoGebra.

CCS CONCEPTS

• **Software and its engineering → Software design tradeoffs**; *Maintaining software*; *Software verification and validation*;

KEYWORDS

design quality, technical debt, design debt, debt prioritization

ACM Reference Format:
Reinhold Plösch, Johannes Bräuer, Matthias Saft, and Christian Körner. 2018. Design Debt Prioritization: A Design Best Practice-Based Approach. In *TechDebt '18: TechDebt '18: International Conference on Technical Debt , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3194164.3194172

TechDebt '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5713-5/18/05…$15.00
https://doi.org/10.1145/3194164.3194172

1 INTRODUCTION

The technical debt (TD) metaphor in the context of software was first introduced by Ward Cunningham in 1992 [8]. For the last 7-10 years, it has been rising in popularity within the software development community and is used as a mechanism to label and communicate (internal) software quality issues, hidden risks for future development and maintenance, as well as the costs incurred. Doing things in software development "quick and dirty" (which is often a deliberate decision) is like incurring debt that causes interest payments in terms of extra efforts to be spent in the future (similar to financial debt). Consequently, technical debt (TD) expresses the effort necessary to clean up software as well as the extra costs.

According to [17], TD can be divided into ten coarse-grained TD types: requirement, architectural, design, code, test, build, documentation, infrastructure, versioning, and defect debt. In the literature, the type that gains most attention is code debt, followed by architectural, test, and design debt [2, 17]. A similar picture of the different TD types is given by [1], with the one exception that code debt is not as dominant in the literature according to their findings. In this work, we narrow down the focus of TD to design debt based on our previous work in this research area and the identified lack of literature. For the sake of clarification, we rely on the definition of design debt as mentioned in [2]. There, the authors refer to the violation of design principles in respect of design debt and classify it in between code and architectural debt.

To deal with existing debt in a software system or to prevent potential debt from being introduced, Li et al. [17] identified and propose eight TD management activities. These eight activities are the identification, measurement, prioritization, prevention, monitoring, repayment, documentation, and communication of TD. From this set, identification (detection of TD by using techniques such as static code analysis), measurement (quantification of TD by using estimation techniques), and repayment (resolving TD by techniques such as re-engineering or refactoring) receive the most attention in scientific literature with support of appropriate tools and approaches [17]. Consequently, there is a lack of research for other activities but an identified need from practitioners and industry. For instance, prioritizing (ranking of identified TD according to pre-defined rules) is mostly based on hunches, experience, and knowledge regarding the code base, but without method support and exact numerical values [28].

This work contributes especially to the activity of prioritizing TD, in particular design debt, by identifying and measuring violations of design best practices. These design best practices were systematically derived from fundamental design principles such as information hiding, single responsibility, or open-closed principle, and were empirically examined regarding their importance [4]. Besides, we developed a tool called MUSE that can identify the design best practice violations directly in the source code for projects written in Java, C#, and C++ [23]. This tool, therefore, contributes to the identification and measurement of TD. The result of a MUSE application is a list of violations of design best practices that is transferred into a portfolio matrix, which is used to prioritize identified issues and to communicate design debt (visualization of TD to discuss and manage it appropriately) to the developers and stakeholders.

The goal of this work is to propose and discuss a method for prioritizing and communicating design debt. To illustrate this method by means of a case study, this work uses the open source project GeoGebra.[1] By means of the case study, we also touch upon the management activities' identification and measurement but do not focus on them.

The remainder of the paper is structured as follows: Section 2 presents and discusses related work in the context of identification, measurement, prioritization, and communication of design debt. Section 3 presents our approach for identifying and measuring design debt based on violated design best practices and our benchmarking-based approach. Section 4 presents our prioritization approach and Section 5 applies this approach to the open-source project GeoGebra. Section 6 discusses the benefits and limitations of our approach and Section 7 draws some conclusions and presents ideas for future work.

2 RELATED WORK

As mentioned in the introduction, this work addresses the TD activities of identifying, measuring, prioritizing, and communicating design debt. Thus, this section provides a state-of-the-art overview of these activities based on the mapping study of Li et al. [17] and extended by recent approaches.

2.1 Design Debt Identification

To identify design debt, a set of approaches has emerged from the research field of measuring and assessing design quality. For example, in [13] the authors propose a metric-based approach that relies on simple metrics such as lines of code or number of bug reports, as well as more complex metrics such as lack of cohesion. Regardless of which metric set is used, each metric is accompanied with threshold values that define the acceptance range of the metric and help to identify design debt.

While considerable effort has been undertaken to increase the accuracy of metric-based measurement results in the context of TD [14], the general issues of using metrics to express design aspects were already criticized in [18]. Thus, instead of using metrics, which are too fine-grained for design concerns when used in isolation, the approach of identifying (code and) design smells has emerged [18]. A design smell that embodies a symptom of the software design is a good indicator of parts that should be refactored. For instance, a god class – as one example of a design smell – may reveal a class that is overloaded with functionality and should be divided into smaller and more appropriate abstractions. An approach for detecting design smells in the context of design debt is shown for eight design smells in [19] and for two smells in [10].

Another group of researchers pursued the idea of using design smells as an indicator for design debt [30]. In their particular study, they concentrated on the design smell of a god class and identified the symptoms in two software systems. Their results show that god classes are more frequently changed. Consequently, they contain more design debt than other classes and are worth monitoring and managing [30]. Based on these findings, the group enhanced their approach and incorporated, for instance, the identification of rule violations from the static code analysis tool FindBugs into their design debt identification approach [31, 32]. As a conclusion, this provides a more elaborate view on design debt.

The idea of using rule violations of static code analysis tools has also been applied to other approaches, which mainly focus on code debt but are transferrable to design concerns. Software Quality Assessment based on Lifecycle Expectations (SQALE), for instance, provides a TD calculation model that includes three abstraction levels for defining quality: characteristics, sub-characteristics, and (rule-based) requirements [15]. Thus, the lowest abstraction level represents the rules (from a static code analysis tool) that identify the design debt through non-compliance directly in the source code [16]. A second approach was recently published in the Object Management Group (OMG) standard for automated TD measure [11]. The foundations of this standard are 86 source code measures (rules) for the quality characteristics maintainability, reliability, security, and performance efficiency [27]. Lastly, the Software Improvement Group (SIG) proposed a TD model, which uses the source code analysis tool Software Analysis Tool (SAT) for identifying TD in source code [21].

2.2 Design Debt Measurement

After the identification of design debt through methods mentioned before, it is necessary to quantify the remediation effort by using appropriate estimation techniques. For assessing metric-based measures, [13] proposes interpretation functions. These functions take the input value and map it to an interval between 0 and 1, where 0 means no occurrence and 1 represents the maximum occurrence of the issue. The most difficult step in defining these interpretation functions is the definition of the thresholds that specify the gate for 0 and 1 [13]. The authors in [13] propose relying on historical data and experience from earlier projects. Despite the threshold issue, this approach does not propose a method to aggregate the result of interpretation functions to a value expressing design debt.

In contrast, [19] shows a design debt measurement approach that operationalizes the impact of identified design smells, which are then aggregated to a single value called the Debt Symptoms Index (DSI). In more detail, the impact of a design smell is characterized by three factors. The first factor is the influence, which expresses how strongly a design smell affects the criteria of good design according to [7]. To evaluate this influence, seven senior software designers were interviewed. The second factor is the granularity of the design

[1] https://www.geogebra.org/

smell referring to the design entities (e.g., class and method) that it affects. Finally, the third factor is the severity of each identified design smell measured by one or more metrics. By combining these three factors for each identified design smell, a Flaw Impact Score (FIS) can be calculated. Finally, adding up all impact scores and normalizing this sum by the lines of code of the system returns the DSI.

While the DSI quantifies design debt in a single value that can be monitored over multiple releases, the value fails to express the actual effort or financial cost associated with the identified debt [19]. In contrast, the SQALE approach calculates the total of the required effort to fix all violations for each rule in a straightforward way [15]. The mechanism to calculate the effort to fix a violation can be configured by either a linear function for increasing effort with an optional offset (e.g., the time required only once to understand a specific rule) or a constant flat rate for fixing any violations of a rule. The resulting TD is given in person-days and computed as follows.

$$RC = \frac{\sum_{rules} effortToFix(violations_{rule})}{8[hr/day]}$$

Similar to SQALE, the OMG standard proposes an aggregation model to predict future corrective maintenance costs based on the identified rule violations. Therefore, the standard provides a default remediation effort (and effort range) for fixing the occurrence of a particular rule violation [11]. These values were derived from interviews with developers from several organizations.

SIG uses another approach compared with the aforementioned methods. Instead of relying on absolute effort estimations, each debt item is evaluated by comparing it with a benchmark base (of 44 systems) [21]. If it is within the top 5% of the products in the benchmark base, the respective item receives a 5-star rating. If it is in the next 30%, it receives a 4-star rating, and so on. If it is in the worst 5% of the products, it receives only a single star. Based on the star rating of a software product (as-is state), the effort to increase the quality of the product to the ideal quality level is estimated.

Due to the lack of further design debt measurement approaches, this discussion slightly touches upon the field of code debt for additional ideas. The CAST approach, for instance, proposes a model that relies on coding violations similar to the SQALE approach [9]. Before these violations are used as a source for the calculation of TD, they are grouped into low, medium, and high severity categories. The CAST approach assumes that only a fraction of the violations is required to be repaired in the future in order to keep the application productive, i.e., 50% of the high severity violations, 25% of medium violations, and 10% of low severity violations. Regardless of the severity, addressing a single violation is assumed to take one hour and cost $75 on average. In conclusion, the CAST model calculates code debt using the following formula:

$$effortToFix(violations) = \begin{cases} violations * effortPerViolation + offset, Function = linear \\ constantEffort, Function = constant \end{cases}$$

$$TD = (50\%\, high\, severity + 25\%\, medium\, severity + 10\%\, low\, severity) * 1\,[hr\ to\ fix\ a\ single\ violation] * \$75\,[cost/hr]$$

The model has been applied in practice to several hundreds of software projects of different industry segments, sizes, and technologies. Based on the data of these applications, the authors state that each of the investigated projects is burdened with an average code debt of $3.61 per lines of code [25]. However, in [9] the authors experienced other values for TD when using alternative estimates as values for the parameters in their formula. They calculated an average of $15.62, which is considerably different from the originally proposed value.

2.3 Design Debt Prioritization

With a design debt measurement, the effort of remediation actions is quantified and can be used for decisions about improvement actions. However, developers are left in the dark as to where to start addressing the debt and which identified issues are worth it. Consequently, the next TD management activity concentrates on prioritizing identified design debt. Therefore, it is required to consider debt from a business value perspective. This view is expressed by non-remediation costs representing the negative consequences on the business value of a project due to remaining debt.

One of the approaches that supports this activity was proposed by [29]. There, the authors continue the idea of identifying and measuring design smells – in particular, god classes – in the source code, which are evaluated according to their refactoring cost and the quality gained from the refactoring. To identify the god class, [29] utilize the detection strategy for a god class as proposed by [18]. This detection strategy is a combination of three object-oriented metrics including concrete threshold values. In the course of the prioritization approach, this detection strategy also defines the cost of paying back design debt because the authors argue that classes that are slightly outside the thresholds will be easier to refactor than classes that have multiple magnitudes outside the accepted gates.

The second dimension, which represents the impact of debt on quality, is determined by the quality characteristics of correctness and maintainability [29]. While [29] operationalize correctness by calculating the defect likelihood, maintainability is measured by the changing likelihood of an identified god class. The combination of these two characteristics returns an assessment of the quality enhancement gained from refactoring a particular god class.

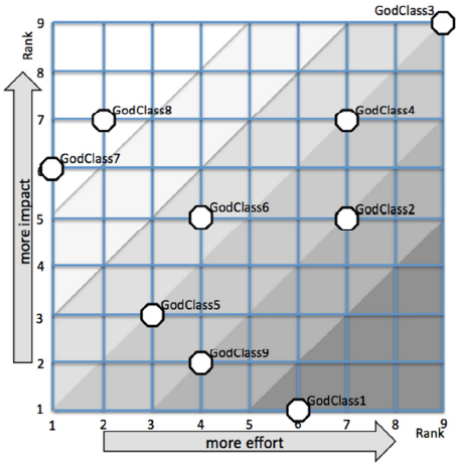

Figure 1: Prioritization of God Classes, from [29]

Given these two assessments of each god class, the prioritization approach continues with a combination and visualization thereof. The result of this step is a prioritization matrix as depicted in Figure 1. According to the arrangement of the identified issues in the matrix, concrete remediation actions can be derived. In more detail, development resources should be invested in those god classes that are on or above the diagonal [29]. The reason, therefore, is that they tend to have a balanced effort/impact ranking or even a higher quality gain in relation to the required effort.

An improvement of the SQALE method (v1.0, see [16]) includes a business impact estimation model that represents the business perspective of TD. It is used to prioritize the remediation actions for the non-conformities of those issues that have the highest/best return ratio. For this, SQALE considers different types of non-remediation (see Table 1) with an associated factor for each type to quantify the penalty for each violation of the respective type.

To calculate the so-called business impact index, it is necessary to sum up all non-remediation factors with a given scope. This quantifies the business impact of the identified findings (i.e., rule violations - see Section 2.2) made on the code. Based on this index and the TD assessment from the measurement approach, the SQALE approach arranges the identified debt within a two-dimensional field, as shown in Figure 2. The interpretation of this graph is similar to the previous approach because the issues with the lowest debt but highest business impact are in the top left corner. In other words, the higher the slope of each item (as shown by the shaded area), the more lucrative the investment in it [16].

In the further course of this work, we present a prioritization approach that is based on portfolio techniques, although another group of researchers also use the portfolio term in this context [12]. To be more concrete, the group proposes a portfolio approach for TD management, which addresses the goals of prioritization but from a more general point of view [12]. Thus, they pursue the idea of characterizing each TD item (design, defect, test, or documentation debt) by a defined template, which requires the estimation of the cost (principal), scope (interest amount), and change likelihood (interest probability) [26]. Based on these characterizations, the project release planning should then consider those TD items with the highest scope and change likelihood as well as the lowest cost estimation.

Table 1: Sample of Non-Remediation Factors [16]

NC Type	Description	Type Sample	Non-Rem. Factor
Blocking	Will or may result in a bug	Division by zero	5,000
High	Will have a high/direct impact on the maintainance cost	Copy and paste	250
Medium	Will have a medium/potential impact on the maintainance cost	Complex logic	50
Low	Will have a low impact on the maintainance cost	Naming convention	15
Report	Very low impact, it is just a remediation cost report	Presentation issue	2

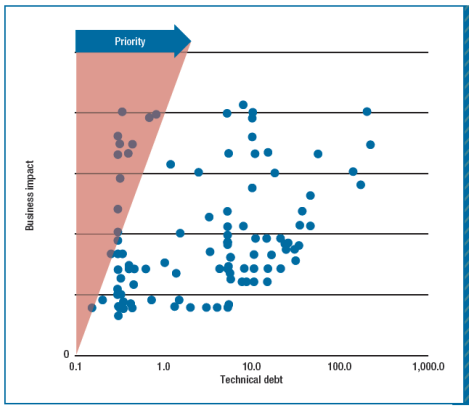

Figure 2: Prioritization based on SQALE method, from [16]

2.4 Design Debt Communication

The last TD activity considered in this section is the communication of design debt. According to the mapping study of Li et al. [17], there is just one work available that addresses this task. This is the SQALE method that tries to express the business impact of TD, which facilitates the communication to stakeholders, who might not be as technically involved in the software system compared with developers. Besides, decisions for additional development and maintenance resources are easier to find and to manage based on a business impact perspective.

3 DESIGN DEBT IDENTIFICATION AND MEASUREMENT

This section presents the techniques we pursue when identifying and measuring design debt.

3.1 Identification of Design Best Practice Violations

In previous work, we focused on the general aim of measuring, assessing, and improving the design quality of an object-oriented software system [22]. Although there are different approaches available that try to achieve the same goal, we identified a lack of proper guidance for software developers and designers throughout the design improvement process. Consequently, we proposed the novel idea of operationalizing the compliance of source codes with fundamental design principles such as the information hiding, single responsibility, or don't repeat yourself principle. However, due to the problem that these principles are still too vague to be measured, more tangible design best practices were systematically derived from the principle descriptions. Finally, the relationship between the design principles and design best practices were transferred into a design quality model [22], which was subject to empirical investigations [5].

While the design quality model could be used to discuss design debt on the level of design principles or even design quality attributes, this work narrows down the focus to the level of design best practices. An example of such a design best practice (design heuristic according to [24]) is *AvoidPublicFields*, which negatively affects information hiding when violated [5]. The main advantage of design best practices is that they are concrete enough to be properly applied by software developers and their adherence can be verified by using static code analysis.

To automate the identification of design best practice violations, we developed a tool called MUSE that currently implements 67 design best practices for the three object-oriented programming languages C++, C#, and Java; for the latter, 49 out of the 67 are applicable [23]. Whereas the entire tool chain of MUSE is explained in [23], the final result of its execution on a project is a list of identified design best practice violations, including the number of checked entities (e.g., no. of classes, methods, or fields). Besides, each violation is linked to the source code to easily guide software developers to the origin of the design debt.

3.2 Benchmarking-Oriented Measurement

After the identification of design best practice violations, the person who investigates the results has to figure out whether the number of violations is acceptable or represents debt that needs to be paid off. Based on our experience in assessing code quality and design quality ([20] and [6], respectively), we propose an assessment approach that is based on the benchmarking concept. In general, benchmarking is a well-established concept in many business areas where (similar) objects (e.g., products, services, processes, organizations) are checked against each other for specific purposes, for instance, the evaluation of the value of an object or to derive suggestions for improvement. In the context of this work, we apply this approach to the judgment of design debt.

Therefore, it is necessary to build a benchmark database that contains the quality data of all reference projects. In this context, it is adequate to store the number of design best practice violations for each practice and for each reference project. In addition, we store values of various size metrics for each project that are needed for normalizing the absolute values. As mentioned before, MUSE provides this data, which are computed for each reference project and then stored in the benchmark database.

Based on the benchmark database, a benchmark suite has to be built that is tailored to a specific application context. In other words, the suite comprises only those projects that are selected as adequate reference projects. Afterwards, we calculate a value distribution for each measure on the basis of their normalized values. As a result, these value distributions reflect commonly used statistical figures such as quartiles quintiles, deciles, or percentiles. Figure 3 depicts a quartile like value distribution and divides the normalized values of the measured reference projects into four areas, Q_1-area to Q_4-area, which are delimitated with the minimum and maximum values of the benchmark suite projects, respectively. If an investigated project is below the benchmark minimum, it is in the additional Q_0-area. Vice versa, this also applies to the maximum value. A project is classified within the Q_5-area if it exceeds the maximum.

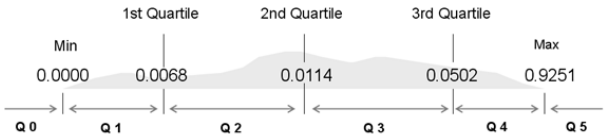

Figure 3: Benchmark distribution in quartiles

Given this value distribution for each design best practice, the number of design best practice violations of the investigated project can be checked against the given distribution for retrieving the achieved quartile. If, for instance, a design best practice has 300 violations in 500 classes and the value distribution of the best practice is like Figure 3, the normalized value (300/500 = 0.6) is between 0.0502 and 0.9251, resulting in a quality index of Q_4. In this particular example, Q_4 indicates a comparison with the worst 25% of projects in the benchmark suite (25% of projects with a very high number of violations of this design best practice).

The result of the benchmarking-oriented measurement approach is a judgment of the design debt by the achieved quality quartile for each design best practice of the investigated project. This already provides an indicator for prioritizing design debt, when the project aims at a target quartile level. In other words, the number of violations of those design best practices that are higher than the target quartile needs to be reduced. Although this addresses the TD activity of prioritization, we consider prioritization from a more elaborate view, as shown next.

4 DESIGN DEBT PRIORITIZATION AND COMMUNICATION

The identification and measurement of design debt are important activities for design debt management. However, without properly prioritizing and communicating design flaws, it is difficult to address the debt with the highest return on investment. This section shows the approach we propose for these activities.

4.1 Importance of Design Best Pratices

While the benchmark-oriented measurement approach derives a quality level (quality index) for each design best practice, it is unclear whether the design best practice is important to achieve good object-oriented design. We disclosed this uncertainty by conducting an online survey among 214 participants, who had to judge the importance of a set of design best practices [4]. For the sake of completeness, we narrowed down the focus of the online survey to Java-related best practices due to the concentration on Java-based systems in our evaluations. Nevertheless, from the survey result we derived a default importance and a standard deviation for each Java design best practice. Based on the importance assessment and a quality index for each design best practice of the investigated project, prioritization tasks are supported by these two viewpoints.

4.2 Portfolio-Based Prioritzation

For properly communicating a prioritization result, we propose a portfolio-based assessment approach, as briefly introduced in [6]. This approach combines the importance with the quality index and provides an arrangement of the measured design best practices

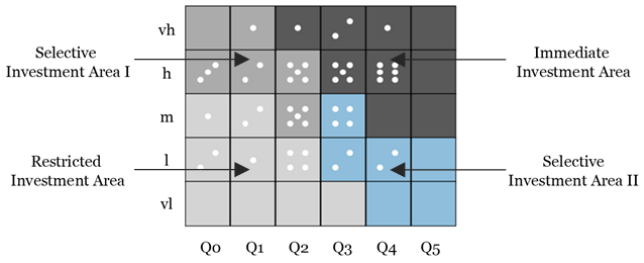

Figure 4: Investment areas of portfolio matrix

within a portfolio matrix. Figure 4 depicts this matrix for the measurement result of a particular project; in total, the 49 design best practices for Java are presented. Taking care of all 49 best practices is time expensive and could be overwhelming for the project team. Thus, the portfolio-based assessment approach groups the design best practices into four so-called investment areas highlighted in different colors.

The dark gray area is the immediate investment area, which contains design best practices that have high design debt and are considered as (very) important. Consequently, investing improvement effort into these design best practices returns the highest quality gain. Next, the light gray and blue areas are called selective investment areas, meaning that they are worth investing in design enhancements based on selective criteria. Hence, it is recommended to choose those design best practices that are relevant to the project without over-engineering different parts or investing in unimportant design aspects. Lastly, the very light gray area in the bottom left corner is the restricted investment area that contains design best practices with good quality and low importance. Investing effort into this corner is not recommended.

This conceptual framework of the portfolio-based assessment approach was discussed with senior developers from industry and characterized by using a feasibility study [3, 6]. These explorative investigations returned a very positive feedback with regard to narrowing down the measurement result and to directing the focus of design discussions to those that are in the immediate investment area. In these discussions, we proposed an aggressive improvement strategy, meaning that the overall design goal is to have no violations in the immediate investment area. While some developers agreed in achieving this goal, others would pursue a more selective and conservative strategy.

4.3 Overall Design Debt Prioritization Approach

Considering the presentations in Sections 4.1 and 4.2, we propose the following design debt prioritization approach:

1. *Define the importance of design best practices:* For this purpose, projects can rely on the default importance derived from our survey on the importance of design best practices [5]. In case this default importance does not reflect the quality needs of the projects, derivations of this default importance are acceptable as long as they remain within the defined boundaries of our survey findings [4].
2. *Define investment strategy:* The investment strategy defines the quality goals using the portfolio matrix without being distracted by the possible costs of fixing the revealed design debt. The proposed default strategy is to move all design best practices out of the immediate investment area. A less offensive strategy could be to move only design best practices with a very high importance out of the immediate investment area. Another strategy could be to move all design best practices (regardless of their importance) to the next quartile.
3. *Consider business needs:* It might not be reasonable to follow the selected investment strategy for the entire product but to differentiate on the component level. Typically, different components have different importance levels from the perspective of the business value. The investment strategy should be aligned with these business needs. Sometimes, business needs are also reflected in technology roadmaps that define which components are re-engineered in the future. Investing in components that are to be re-engineered in the near future definitely needs a different investment strategy than components that are already aligned with the technology goals of the product or project.

For the sake of simplicity, we presented our portfolio approach on the level of an entire project. In reality, a more differentiated view with different investment strategies on the component level might be necessary to meet the business needs. Nevertheless, this does not falsify our improvement approach but just makes clear that these different needs can be easily fulfilled.

Based on our current experience from previous investigations, it can be summarized that the proposed approach leads a project team to a design quality assessment – represented by the portfolio-matrix – in a gradually and jointly manner. Further, the retrieved assessment incorporates the opinions of involved stakeholders and can be easily communicated to others. For instance, we observed that an external person that may not be part of the development process can intuitively grasp the quality state of the product to judge whether it is worth to invest effort in paying off debt.

5 APPLICATION OF THE APPROACH

In the further course of this work, we present our implementation of the design debt management activities with respect to an open-source project. Besides, we discuss and provide suggestions for prioritization strategies that go beyond the previously proposed ideas and consider, for instance, the business needs of the project.

5.1 Study Object

The study project of this work is the GeoGebra project, which is one of the leading open source software products for teaching MINT subjects in particular mathematics. The investigated revision of GeoGebra is revision number 57818, as downloaded from the open-source repository on 14th November, 2017. When considering the amount of Java source code, this revision has 844,066 logical lines of code. Further, a component-based analysis of the project shows that it consists in its core of a web and desktop component, both building on a common kernel component.

5.2 Design Debt Measurement and Prioritization

To identify design debt in GeoGebra, we executed MUSE containing the 49 design best practices for Java. This execution derived the violations of each design best practice as well as entity sizes for normalizing the number of violations. These normalized values were then checked against a benchmark suite to derive the quality index as described in Section 3. For the sake of clarification, we pre-defined the benchmark suite and relied on 50 reference projects that amount to 4.55 million logical lines of code and have proven to be solid for this type of investigation (see benchmark suite defined in the Appendix of [6]).

In order to define the importance of each design best practice, we relied on the findings from the conducted survey [4]. Finally, we mapped the design best practices into our portfolio-matrix to provide a more elaborate view of the measurement result (80,717 violations in total), as shown in Figure 5. Accordingly, GeoGebra has quite some design debt, as highlighted by the 26 design best practices in the immediate investment area, of which 19 practices fall in Q_4. In total, these 26 best practices are responsible for 47,108 violations.

Considering all violations as equal is unfair because they differ in their severity and in regard to the design entity they operationalize. For instance, the design best practice *AvoidPackageCycles* identifies cycles on package level and requires a restructuring of the packages to break up the cycle. In contrast, *AvoidPublicFields* is a practice that works on the level of class fields and forces a reconsideration of their visibility. To differ between the various design best practices, the design quality model (as mentioned in Section 3.1) specifies the assignment of design best practices to product parts. The product parts affected by the 49 design best practices are listed in Table 2.

While issues related to methods, for instance, are less time consuming to fix compared with package problems, we assigned different remediation costs to each product part. In other words, we assume that fixing a violation with respect to a class violation takes ten minutes compared with a package issue that takes one hour. These remediation costs can be used as suggested but can, of course, be replaced by other values derived, for example, from experience gained with refactoring in the past. Despite the circumstance that we will use the assumed remediation costs throughout the paper, it is important to mention that their values do not impact the underlying concept of the approach, which is in focus of the work.

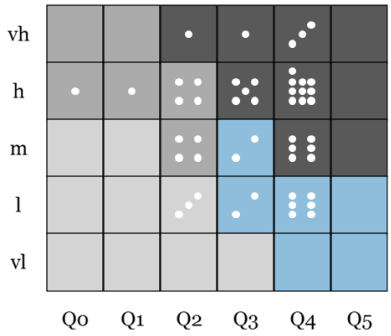

Figure 5: Portfolio matrix of design debt measurement

Table 2: Assumed Remediation Costs

Product part	Remediation cost in minutes
Interface	3
Method	5
Class	10
Type	10
Source Code	20
Package	60
Component	60

Based on these assumptions and the assignment of design best practices to product parts, it is possible to calculate the total amount of required time effort. For GeoGebra and the identified design debt of 80,717 violations, the effort would accumulate to 9.55 person-years when assessing a year with 1,600 working hours. Investing this much effort for removing debt that may have a negative impact on quality attributes (e.g., maintainability, understandability, or functional suitability) is inefficient for the business needs of the product. Hence, our first recommendation for a more focused design debt prioritization is to narrow down to those design best practice violations that are in the immediate investment area.

5.3 Improvement Strategy Consideration

As previously mentioned, 26 design best practices are in the immediate investment area, which is visualized by the dark grey shading in Figure 5. Accordingly, an aggressive investment strategy could define the quality goal of having no design best practice in this area. Therefore, it is necessary to move, for instance, a design best practice from field Q_4/vh to Q_1/vh or a practice from Q_4/m to Q_3/m. In other words, this improvement strategy does not force an elimination of all violations but rather the number of violations that is necessary to achieve one of the selective investment areas. Table 3 provides an overview of the 26 design best practices and depicts the number of fixed violations required to achieve the quality goal.

According to this table, 26,271 violations must be addressed to move all design best practices out of the immediate investment area. This number represents a reduction of the total number of violations by 32.5%. In considering the impacted product factors of each violation and by using the assumed remediation cost from Table 2, these violations cause a maintenance effort of 3.84 person-years. While the consideration of the aggressive improvement strategy reduces the workload by 40.2%, this is still a high effort to invest. Consequently, it is necessary to go one step further and to focus concrete design debt mitigation actions on those items that are worth improving from a business perspective.

5.4 Business Need Consideration

When talking about design debt, we argue that the consideration of business needs is important to avoid investing resources into product components that are end of life or will be replaced by newer technologies. Valuable hints for these business needs are

Table 3: Design Debt Measurement

Design best practice	No. of violations	No. of required fixes
AvoidDuplicates	4,482	2,207
AvoidUsingSubtypesInSupertypes	49	36
AvoidPackageCycles	2,562	1,981
AvoidCommandsInQueryMethods	1,223	956
AvoidPublicFields	743	733
DocumentInterfaces	2,619	1,296
AvoidLongParameterLists	374	246
UseInterfacesIfPossible	17,653	11,339
AvoidStronglyCoupledPackages	115	36
DontReturnUninvolvedDataFromCommands	335	31
UseCompositionNotInheritance	275	176
DocumentPublicClasses	1,012	518
AvoidPublicStaticFields	62	39
AvoidDiamondInheritanceStructuresInterfaces	461	316
AvoidLongMethods	1,373	744
AvoidSimilarNamesForDifferentDesignElements	48	48
AvoidUnusedAbstractions	123	15
CheckUnsuitableFunctionalityOfClass	182	72
AvoidSimilarAbstractions	24	24
UseInterfacesAsReturnType	5,301	3,363
AvoidSimilarNamesForSameDesignElements	696	439
AvoidRepetitionOfPackageNamesOnAPath	3	3
AvoidRuntimeTypeIdentification	5,268	1,189
AvoidDirectObjectInstantiations	939	182
CheckUnusedSupertypes	574	274
AvoidMassiveCommentsInCode	612	8
Sum:	47,108	26,271

product roadmaps and development plans (see also Section 4.3). They contain the information of up-coming features or obsolete components due to changing customer requirements. Furthermore, the identification of those components that show the most development activities is another source of interest for prioritizing design enhancement actions.

In the context of the GeoGebra project, we applied the idea of concentrating improvement actions on the component that is currently most actively enhanced. Therefore, we analyzed the last 400 commits on the source code repository conducted within the last two weeks (up until 14th November). In more detail, we analyzed changes in the source code from revision 57418 to 57818 and identified the component with the most delete, add, and modify file actions. From 460 deleted, added, or modified files within the last 400 commits, 146 files from the /geogebra/web component were affected. In relation to the size of this component and compared with the others, /geogebra/web was identified as the part of the project with the most development activities.

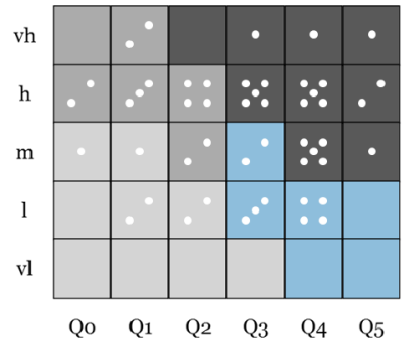

Figure 6: Portfolio matrix of design debt measurement for web package

Based on this concentration to the web component, a re-execution of MUSE returns a slightly different design debt picture. In more detail and as shown in Figure 6, this component has a lower design debt than the entire GeoGebra project because the number of design best practices in the immediate investment area decreases from 26 to 21. Besides, for three design best practices the component is performing even better than the entire benchmark suite, as depicted by the practices in column Q_0. In contrast, four design

Table 4: Design Debt Measurement for Web Package

Design best practice	No. of violations	No. of required fixes
AvoidPackageCycles	561	491
AvoidCommandsInQueryMethods	116	88
AvoidPublicFields	28	27
DocumentInterfaces	1,090	782
UseInterfacesIfPossible	1,410	924
AvoidNonCohesivePackages	13	1
UseCompositionNotInheritance	102	77
DocumentPublicClasses	209	151
AvoidPublicStaticFields	4	2
AvoidDiamondInheritanceStructuresInterfaces	169	143
AvoidLongMethods	117	24
AvoidSimilarNamesForDifferentDesignElements	4	4
AvoidUnusedAbstractions	21	9
CheckUnsuitableFunctionalityOfClass	21	9
UseInterfacesAsReturnType	553	305
AvoidSimilarNamesForSameDesignElements	32	9
AvoidRepetitionOfPackageNamesOnAPath	1	1
CheckUnusedSupertypes	99	57
AbstractPackagesShouldNotDependOnOtherPkg.	14	3
DontReturnMutableCollectionsOrArrays	33	6
AvoidMassiveCommentsInCode	107	18
Sum:	8,651	3,131

best practices move to Q_5, meaning that the normalized number of violations is worse than the highest benchmark value. In particular, these are the design best practices *AvoidPackageCycles, UseCompositionNotInheritance, AvoidDiamondInheritanceStructuresInterfaces,* and *CheckUnusedSupertypes*.

Regardless of the detailed design debt changes, the improvement strategy recommends an investment in the dark grey area to move all practices out of it. Therefore, Table 4 highlights the number of violations that need to be fixed for each design best practice. In total, there are 3,131 violations representing 3.88% of all identified design issues in the GeoGebra project. From the viewpoint of maintenance effort in workload, the remaining list of violations accumulates 0.52 person-years. Communicating this debt value to a stakeholder is convincing, as it addresses the most relevant design debt issues (high importance and bad quality state) and considers the business need, identified by the component with the highest development activities.

6 DISCUSSION

The presented design debt prioritization approach has benefits for practitioners but comes with some limitations. Both sides will be summarized and discussed below.

6.1 Benefits

Design best practices are the items measured by the approach. To be more specific, design debt is operationalized by the violations and non-compliance of these design best practices in the source code. The list of practices is not a set of randomly selected guidelines, but rather is systematically derived from fundamental design principles and put in relation by using a formal design quality model [22]. Besides, the role of these design best practices in the context of design principles, which positively impact design quality, is further examined with 31 senior software developers divided into six focus groups [5]. For an unrelated design principle investigation of the practices, an online survey among 214 participants empirically determined their individual importance [4]. In conclusion, the approach can beneficially rely on well-investigated practices that are (1) important for object-oriented design, (2) concrete enough to be followed by practitioners (a circumstance that is difficult to achieve when focusing on design principles), and (3) specific enough to be identified by an automated measurement tool.

This approach implements the measurement tool MUSE to identify design best practice violations [23]. For the sake of clarification, the approach is not limited to this tool. In fact, it is possible to apply any static analysis tool, including static code analyzer.

After a measurement, the result is mapped into the portfolio-matrix to determine those design debt issues that are considered to be (very) important but achieve a bad quality state. This step narrows down the measurement result to those design best practices that should be further considered. Moreover, it is easier to communicate real design flaws without getting lost in details that are not worth being discussed. For instance, time is lost when discussing a violation of a practice, which is assessed to be unimportant or already within a good quality state (compared to the benchmark).

Another advantage of the proposed approach is the business-related differentiation of design best practice violations. Taking into account that violations of the same design best practice differ in their business impact, it is important to adjust the approach to the business needs of the project. For example, the analysis can focus on components that will undergo feature extensions. This provides the benefit of concentrating improvement actions in software parts that have a high return on investment and are unlikely to be replaced in the near future.

In conclusion, the approach achieves a focused view on design debt by drilling down the measurement result to the most relevant design debt issues and to product parts with a high business impact.

6.2 Limitations

In the presented approach, a benchmarking technique is applied to judge the achieved quality state. While the used technique provides weights for achieving a different quality index, building the benchmark base or deriving the benchmark suite are disadvantages of the proposed approach. In fact, identifying relevant reference projects is difficult because projects are usually subject to different constraints, apply different technologies, or simply differ in their size. Consequently, building a benchmark base requires a trade-off in using the most appropriate projects. For a company, this is quite difficult since there is limited access to similar projects in the corresponding domain.

Nevertheless, current trends in software engineering motivate the practical application of a benchmark-based technique since companies pursue a service-oriented development approach with a focus on building small and independent services, which are orchestrated to an application that serves the customer's needs [33]. All these small services – so-called microservices – are deployable projects that could represent the reference projects in the benchmark base. Thus, it would be possible to compare a project with all services developed by the company, a task that was difficult in former times due to the thinking in monolithic systems with little chance of finding comparable subjects.

7 CONCLUSION AND FUTURE WORK

In this work, we presented a comprehensive approach for identifying, measuring, prioritizing, and communicating design debt. The TD management activities of identifying and measuring design debt with our approach are already targeted towards prioritization, as the identification and measuring are based on violated design best practices, which is an advantage compared to metric or smell-based approaches, as all of these practices provide very concrete hints regarding what to improve in the source code.

For the quantification, such as for the measuring of the design debt, we rely on a similar approach as [25], i.e., we quantify the remediation costs for a single violation of a design best practice with a fixed time factor. Other approaches for calculating the remediation costs, e.g. relying on historic data, are easily incorporable in our approach without invalidating the general prioritization approach.

For the TD management of prioritization, we rely on a portfolio matrix with the importance of the design best practice and the achieved quality index (calculated with our benchmarking approach) as the two dimensions of the portfolio. Investment strategies, like moving all violations of design best practices out of the immediate investment area of the portfolio, can be used to prioritize

design debt resolution. To concentrate the prioritization further, business needs can be considered, which might affect the general improvement strategy and will also narrow the focus to selected core components of a system. The portfolio matrix used also helps to support the TD management activity of communicating the design debt.

Applying our approach to the open-source project GeoGebra shows that the prioritization approach works well, as it reduces the effort (for fixing all identified problems) from 9.55 person-years to a reasonable 0.52 person-years. The achieved prioritization seems to be reasonable, although we were not able to validate the suitability of the results with the GeoGebra team.

As future work, we therefore first have to perform a suitability study in a real-word context that focuses on the utility of the portfolio-based assessment approach for deriving design improvement actions. The central question here is whether our portfolio-based TD approach helps in defining the improvement actions.

A minor but still interesting point is the development of a more elaborate model for estimating the remediation costs of design best practices based on more basic refactoring tasks that have to be carried out. This could also help to objectify remediation cost estimations.

REFERENCES

[1] Nicolli S. R. Alves, Thiago S. Mendes, Manoel G. de Mendonça, Rodrigo O. Spínola, Forrest Shull, and Carolyn Seaman. 2016. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology* 70, Supplement C (2016), 100–121. https://doi.org/10.1016/j.infsof.2015.10.008

[2] Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Paris Avgeriou, Pekka Abrahamsson, Antonio Martini, Uwe Zdun, and Kari Systa. 2016. The Perception of Technical Debt in the Embedded Systems Domain: An Industrial Case Study. In *8th IEEE International Workshop on Managing Technical Debt (MTD)*. 9–16. https://doi.org/10.1109/MTD.2016.8

[3] Johannes Bräuer. 2017. *Measuring and Assessing Object-oriented Design Principles*. Dissertation. Johannes Kepler University, Linz, Austria.

[4] Johannes Bräuer, Reinhold Plösch, Matthias Saft, and Christian Körner. 2017. A Survey on the Importance of Object-oriented Design Best Practices. In *Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA) 2017*. Vienna, Austria, 27–34. https://doi.org/10.1109/SEAA.2017.14

[5] Johannes Bräuer, Reinhold Plösch, Matthias Saft, and Christian Körner. 2018. Measuring Object-Oriented Design Principles: The Results of Focus Group-Based Research. *Journal of Systems and Software* (2018). https://doi.org/10.1016/j.jss.2018.03.002

[6] Johannes Bräuer, Matthias Saft, Reinhold Plösch, and Christian Körner. 2017. Improving Object-oriented Design Quality: A Portfolio- and Measurement-based Approach. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement (IWSM Mensura '17)*. ACM, New York, NY, USA, 244–254. https://doi.org/10.1145/3143434.3143454

[7] Peter Coad and Edward Yourdon. 1991. *Object-Oriented Design*. Prentice Hall, London, UK.

[8] Ward Cunningham. 1992. The WyCash Portfolio Management System. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum) (OOPSLA '92)*. ACM, New York, NY, USA, 29–30. https://doi.org/10.1145/157709.157715

[9] Bill Curtis, Jay Sappidi, and Alexandra Szynkarski. 2012. Estimating the Size, Cost, and Types of Technical Debt. In *Proceedings of the Third International Workshop on Managing Technical Debt (MTD '12)*. IEEE Press, Piscataway, NJ, USA, 49–53.

[10] Francesca Arcelli Fontana, Vincenzo Ferme, and Stefano Spinelli. 2012. Investigating the impact of code smells debt on quality code evaluation. In *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE, 15–22. https://doi.org/10.1109/MTD.2012.6225993

[11] Object Management Group. 2017. About the Automated Technical Debt Measure Specification Version 1.0 beta 2. (2017). http://www.omg.org/spec/ATDM/

[12] Yuepu Guo and Carolyn Seaman. 2011. A Portfolio Approach to Technical Debt Management. In *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD '11)*. ACM, New York, NY, USA, 31–34. https://doi.org/10.1145/1985362.1985370

[13] Jeanette Heidenberg and Ivan Porres. 2010. Metrics Functions for Kanban Guards. In *17th IEEE International Conference and Workshops on Engineering of Computer Based Systems*. 306–310. https://doi.org/10.1109/ECBS.2010.43

[14] Clemente Izurieta, Isaac Griffith, Derek Reimanis, and Rachael Luhr. 2013. On the Uncertainty of Technical Debt Measurements. In *Proceedings of the International Conference on Information Science and Applications (ICISA)*. 1–4. https://doi.org/10.1109/ICISA.2013.6579461

[15] Jean-Louis Letouzey. 2012. The SQALE method for evaluating Technical Debt. In *Proceedings of the Third International Workshop on Managing Technical Debt*. 31–36. https://doi.org/10.1109/MTD.2012.6225997

[16] Jean-Louis Letouzey and Michel Ilkiewicz. 2012. Managing Technical Debt with the SQALE Method. *IEEE Software* 29, 6 (2012), 44–51. https://doi.org/10.1109/MS.2012.129

[17] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 101, Supplement C (2015), 193–220. https://doi.org/10.1016/j.jss.2014.12.027

[18] Radu Marinescu. 2004. Detection strategies: metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance*. IEEE, 350–359. https://doi.org/10.1109/ICSM.2004.1357820

[19] Radu Marinescu. 2012. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development* 56, 5 (2012), 9:1–9:13. https://doi.org/10.1147/JRD.2012.2204512

[20] Alois Mayr, Reinhold Plösch, and Christian Körner. 2014. A Benchmarking-Based Model for Technical Debt Calculation. In *Proceedings of the 14th International Conference on Quality Software (QSIC 2014)*. IEEE, Dallas, Texas, 305–314. https://doi.org/10.1109/QSIC.2014.35

[21] Ariadi Nugroho, Joost Visser, and Tobias Kuipers. 2011. An Empirical Model of Technical Debt and Interest. In *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD '11)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/1985362.1985364

[22] Reinhold Plösch, Johannes Bräuer, Christian Körner, and Matthias Saft. 2016. Measuring, Assessing and Improving Software Quality based on Object-Oriented Design Principles. *Open Computer Science* 6, 1 (2016). https://doi.org/10.1515/comp-2016-0016

[23] Reinhold Plösch, Johannes Bräuer, Christian Körner, and Matthias Saft. 2016. MUSE - Framework for Measuring Object-Oriented Design. *Journal of Object Technology* 15, 4 (2016), 2:1–29. https://doi.org/10.5381/jot.2016.15.4.a2

[24] Arthur J. Riel. 1996. *Object-Oriented Design Heuristics* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[25] J. Sappidi, Bill Curtis, and Alexandra Szynkarski. 2011. *The CRASH Report - 2011/12 Summary of Key Findings*. Technical Report.

[26] Carolyn Seaman and Yuepu Guo. 2011. Measuring and Monitoring Technical Debt. Advances in Computers, Vol. 82. Elsevier, 25 – 46. https://doi.org/10.1016/B978-0-12-385512-1.00002-5

[27] Richard Mark Soley and Bill Curtis. 2013. The Consortium for IT Software Quality (CISQ). In *Software Quality. Increasing Value in Software and Systems Development*, Dietmar Winkler, Stefan Biffl, and Johannes Bergsmann (Eds.). Number 133 in Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 3–9.

[28] Jesse Yli-Huumo, Andrey Maglyas, and Kari Smolander. 2016. How do software development teams manage technical debt? - An empirical study. *Journal of Systems and Software* 120, Supplement C (2016), 195–218. https://doi.org/10.1016/j.jss.2016.05.018

[29] Nico Zazworka, Carolyn Seaman, and Forrest Shull. 2011. Prioritizing Design Debt Investment Opportunities. In *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD '11)*. ACM, New York, NY, USA, 39–42. https://doi.org/10.1145/1985362.1985372

[30] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the Impact of Design Debt on Software Quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD '11)*. ACM, New York, NY, USA, 17–23. https://doi.org/10.1145/1985362.1985366

[31] Nico Zazworka, Rodrigo O. Spínola, Antonio Vetro', Forrest Shull, and Carolyn Seaman. 2013. A Case Study on Effectively Identifying Technical Debt. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering (EASE '13)*. ACM, New York, NY, USA, 42–47. https://doi.org/10.1145/2460999.2461005

[32] Nico Zazworka, Antonio Vetro', Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman, and Forrest Shull. 2014. Comparing four approaches for technical debt identification. *Software Quality Journal* 22, 3 (2014), 403–426. https://doi.org/10.1007/s11219-013-9200-8

[33] Olaf Zimmermann. 2017. Microservices tenets. *Computer Science - Research and Development* 32, 3-4 (2017), 301–310. https://doi.org/10.1007/s00450-016-0337-0