Technical Debt as an External Software Attribute

Luigi Lavazza
Dip. di Scienze Teoriche e Applicate
Università degli Studi dell'Insubria
Varese, Italy
luigi.lavazza@uninsubria.it

Sandro Morasca
Dip. di Scienze Teoriche e Applicate
Università degli Studi dell'Insubria
Varese, Italy
sandro.morasca@uninsubria.it

Davide Tosi
Dip. di Scienze Teoriche e Applicate
Università degli Studi dell'Insubria
Varese, Italy
davide.tosi@uninsubria.it

ABSTRACT

Background: Technical debt is currently receiving increasing attention from practitioners and researchers. Several metaphors, concepts, and indications concerning technical debt have been introduced, but no agreement exists about a solid definition of technical debt.
Objective: We aim at providing a solid basis to the definition of technical debt and the way it should be quantified.
Method: We view technical debt as a software quality attribute and therefore we use Measurement Theory, the general reference framework for the quantification of attributes, to define technical debt and its characteristics in a rigorous way.
Results: We show that technical debt should be defined as an external software quality attribute. Therefore, it should be quantified via statistical and machine-learning models whose independent variables are internal software quality attributes. Different models may exist, depending on the specific needs and goals of the software product and development environment. Also, technical debt is a multifaceted concept, so different kinds of technical debt exist, related to different quality attributes, such as performance, usability, and maintainability. These different kinds of technical debt should be evaluated individually, so one can better focus on the specific quality issues that need to be addressed.
Conclusions: We show that, to provide it with a rigorous basis, technical debt should be considered and measured as an external software attribute. Researchers and practitioners should build models for technical debt and use them to (1) assess the extent of the technical debt and (2) investigate and assess different ways of modifying software to repay technical debt.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; *Risk management*; *Software evolution*;

KEYWORDS

Technical Debt, Software Quality

ACM Reference Format:
Luigi Lavazza, Sandro Morasca, and Davide Tosi. 2018. Technical Debt as an External Software Attribute. In *TechDebt '18: International Conference on Technical Debt, May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3194164.3194168

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

ACM ISBN 978-1-4503-5713-5/18/05…$15.00
https://doi.org/10.1145/3194164.3194168

1 INTRODUCTION

The term "technical debt" (TD) was originally introduced by Cunningham [6]. Cunningham observes that sometimes "immature" code (i.e., code whose internal quality is sub-optimal) is released to achieve some immediate advantage, e.g., to ship a product in the shortest possible time, and that *"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. [...] The danger occurs when the debt is not repaid"* because *"excess quantities [of immature code] will make a program unmasterable."*

TD has been used as a metaphor in following studies and as guidance for maintenance in some applications (see Section 2). TD can be repaid via refactoring, which can be a costly activity. A cost/benefit analysis should be carried out before starting refactoring, to investigate whether the benefits of refactoring a code section are greater than the costs, and to select the specific code sections related to different TDs whose refactoring will yield the highest returns.

This shows the need to quantify TD, so one can make sound decisions. During development, project managers need to assess the extent of the TD they are taking when deciding whether to code in a suboptimal way now for some short-term benefit and refactor later. During maintenance, maintainers need to identify the code sections that are related to TD and decide which ones need to be refactored and in what order, given the limited amount of resources available for maintenance.

The main goal of this paper is to put the measurement of TD on sound bases. We show how TD fits in the general, commonly accepted framework of Measurement Theory. Specifically, we show that TD is an external software attribute and it should be quantified as such. This also helps clarify the terminology used in the TD research field. We also argue that TD can be defined with respect to other practically relevant software attributes, such as reliability, performance, etc. We show how TD can be computed based on the gap between the actual quality levels of the attributes in software code and the required quality levels.

The remainder of the paper is organized as follows. Section 2 provides background on TD. Section 3 provides the elements of Measurement Theory that are used throughout the paper. Section 4 illustrates the reasons why TD can be viewed as an external attribute of software. Section 5 discusses the role of TD principal and interest. Section 6 discusses the relationships that link TD to several external software attributes. In Section 7, we show that realistic issues concerning TD repayment and software maintenance are coherent with the view of TD as an external attribute described in

the paper. Section 8 draws some conclusions and outlines future work.

2 BACKGROUND: TECHNICAL DEBT

In the last few years, TD has received a good deal of attention from researchers. For example, a recent Systematic Mapping Study on TD and TD management covering publications from 1992 and 2013 detected the existence of 94 primary studies that were used to obtain a comprehensive understanding on the TD concepts and an overview on the current state of research on TD management [20]. The main conclusion of this work was related to the increasing impact TD has on the academia and industry research. Interest, principal, and risk are the main notions used to describe TD, and TD can be categorized into ten categories (such as requirements TD, architectural TD, design TD, code TD, etc.). Most of the studies agreed that TD negatively affects maintainability. As for TD management, some activities—such as TD identification and measurement—received a good deal of attention, while other activities—such as TD representation and documentation—did not receive any attention at all. Finally, a clear lack of tools for managing TD was highlighted: only four tools are available and dedicated to managing TD.

In another Systematic Mapping Study [3], three main research questions where addressed:

(1) What are the types of TD?
(2) What are the strategies proposed to identify TD, which empirical evaluations have been performed, and which artifacts and data sources have been proposed to identify the TD?
(3) What strategies have been proposed for the management of TD, which empirical evaluations have been performed, and which software visualization techniques have been proposed to manage TD?

The findings of the study show that most studies deal with TD at the source code level (i.e. design, defect, code and architecture debt) and researchers detected the existence of TD throughout the entire lifecycle of the project. This implies that ensuring the quality of the project's source code is not the only way to enhance project quality. However, researchers limit the study to the existing problems in the source code. Several studies focus on strategies to manage TD. However, only five strategies (Portfolio Approach, Cost-Benefit Analysis, Analytic Hierarchy Process, Calculation of TD Principal, and Marking of dependencies and Code Issues) were cited and evaluated in more than two papers. Few studies addressed the evolution of TD during the development and maintenance phases of a project.

Nugroho et al. proposed definitions for the debt and its interests [24]. Namely, they define technical debt as *"the cost of repairing quality issues in software systems to achieve an ideal quality level"* and the interests of the debt as *"the extra maintenance cost spent for not achieving the ideal quality level."*

In a recent Dagstuhl Seminar [4] specifically devoted to TD, the following definition of TD was proposed by Avgeriou et al.: *"In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability."* In this case, TD is seen as a set of implementation artifacts that affect negatively maintainability and evolvability.

It can be observed that Nugroho et al. focus on the cost of improving the internal quality of software, while Avgeriou et al. focus on elements of software that cause the internal quality to be low. However, these two aspects are clearly correlated (the lower the quality, the higher the cost of recovering), hence there is no substantial disagreement between the two definitions.

These definitions are a good starting point for approaching the problem of quantifying TD, but they need to be discussed further. For instance, there is not a unique ideal quality level. Consider an object-oriented system where encapsulation has been thoroughly enforced. It is well known that this situation is good for some types of maintenance activities: for instance, changing the implementation of a class to remove a defect will have no impact outside the changed class, thus the scope of the change is limited, and so is the required effort. However, if maintenance is required to improve the efficiency of the system, it is possible that it is necessary to break encapsulation to get direct access to the data of a class: in such case, it is encapsulation itself (an assumed "good" quality) that makes maintenance necessary. In conclusion, there is no ideal quality level: any arrangement of code can result in making a given type of maintenance easier and some other type more difficult.

Ramasubbu et al. deal with TD from a managerial point of view [25]. An organization has to consider that TD can provide advantages (e.g., shorter time to deliver the software product) but with increased maintenance costs. Accordingly, a trade-off has to be devised. To this end, Ramasubbu et al. suggest strategies for dealing with TD, depending on three decision drivers: customer satisfaction needs, required reliability, and the probability of technology disruption.

The SQALE (Software Quality Assessment based on Lifecycle Expectations) method [18, 19] addresses a set of external qualities (like reliability, efficiency, maintainability, etc.). Each of these qualities is associated with a set of requirements concerning internal qualities: for instance, changeability is *"No cyclic dependence," "No public data," "Number of derived classes* < 10*,"* etc. Each of these requirements is provided with a *"remediation function,"* which represents the cost of changing the code so that the requirement is satisfied. This function has to be defined locally by each organization. A *"remediation index"* is then computed for each module, based on the remediation function. Finally, an index is computed for each external quality: for instance, the SQALE maintainability index is computed by adding all remediation indexes linked to all the requirements that contribute to maintainability. This index represents the cost (in work units, in time units, or in monetary units) of satisfying the maintainability requirements.

Overall, the SQALE method provides a reasonable framework to estimate TD. With respect to other methods, it also has the merit of applying the concept to various phases of the development process: e.g., the SQALE testability index addresses the cost of testing that is due to the low quality of the code to be tested. However, there are several issues that limit the value of the proposal: 1) the relationships between external qualities and internal (code) qualities are not supported by any empirical evidence (e.g., no statistically valid models are ever mentioned); 2) thresholds do not appear to be

based on any sound criterion (for instance, why should the number of derived classes be less than 10, instead of, say, 12?); 3) the model appears unnecessarily rigid (for instance, reliability includes testability, but in some contexts one could be interested in a "stand-alone" model of reliability. In addition, there is no guarantee that all the qualities that support testability are actually statistically related to reliability as well). In conclusion, SQALE is a reasonable abstract framework, but a lot of work has to be done to make it a concrete and sound model in a given context.

The OMG (Object Management Group) is working at the specification of Automated TD Measure (ATDM), a measure of TD that can be computed automatically [1]. The proposed measure addresses the TD principal, defined as *"The cost of remediating must-fix problems in production code."* In summary, the measure is a weighted sum of the *"violations of good architectural and coding practices defined in the CISQ Quality Characteristic specifications,"* detected according to the occurrence of specific code patterns. The weight is computed according to the expected remediation effort required for each violation type—as derived from a survey carried out by CISQ—and adjusted based on *"qualification measures,"* which account for technological diversity, complexity, concentration, exposure, and evolution status.

It should be noted that ATDM is a measure of TD that is related to the effort required to repay the debt in *average* conditions. In fact, it is not always the case that a given organization is willing to fix exactly the set of violations identified by the ATDM, or that the cost of fixing each violation will be the one computed by ATDM.

Cunningham suggests also that some sort of compound interest occurs in software development: *"The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation"* [6].

3 BACKGROUND: THE MEASUREMENT OF SOFTWARE ATTRIBUTES

Software artifacts are characterized by a number of attributes, which represents their qualities. We here summarize the concepts related to the measurement of software attributes, which are traditionally divided into "internal" and "external" ones, according to the terminology of [10].

3.1 Internal Software Attributes

Internal software attributes are those attributes of a software artifact that *can be measured* based on the knowledge of the artifact alone. Examples of internal software attributes are: size, which can be measured with Lines of Code (*LOC*) or Number of Methods (*NOM*); structural complexity, which can be measured with Cyclomatic Complexity (*CC*); cohesion, whose lack can be measured with Lack of Cohesion of Methods (*LCOM*); coupling, which can be measured with Afferent (*Ca*) and Efferent (*Ce*) Coupling.

It is relatively easy to compute measures for internal software attributes, i.e., internal software measures. For instance, measuring the size of a class via *NOM* simply requires scanning the code of the class and counting its methods. Several tools are available to compute measures for internal attributes (see for example the list of tools on Wikipedia `https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis`).

Internal software measures are not useful *per se*, e.g., the sheer knowledge of *NOM* of a class does not provide practically useful information to developers. However, an internal software measure can be very useful if it is known that it is related to some practically relevant characteristic of software products, typically an external software attribute, as we explain next in Section 3.2.

3.2 External Software Attributes

External attributes are those attributes that *cannot be measured* based only on the knowledge of the software artifact, but they can be measured taking into account the artifact, its environment, and the interactions between the artifact and the environment. Examples of external attributes are reliability, performance, usability, maintainability, testability, and portability.

Taking reliability as an example, the reliability of a software product depends on the software product itself, but also on its users, and the way the users use the product. For instance, it is to be expected that the reliability of a software product is lower when it is used during testing (i.e., when the product is thoroughly tested) than during operational use. So, the knowledge of the product alone is not sufficient to quantify its reliability. Therefore, it is not possible to quantify the reliability of a software artifact without taking into account its relevant attributes (e.g., its size, structural complexity, and modularity), the relevant attributes of its users (e.g., the degree to which they know the system), and the way they interact with it (e.g., which functionalities are most important to let users attain their "business" goals).

External attributes are in general more difficult to define and quantify than internal ones, since we need to take into account the users and the interactions between the users and the software product, in addition to the software product itself. However, unlike internal ones, external software attributes are relevant to software stakeholders, who are interested in knowing, how reliable, usable, maintainable, etc. a software product is.

For clarity, note that we use the terms "internal" and "external" in this paper according to the reference scientific literature —as in [10], for instance— and not according to the ISO25000 series [11], where they have a different meaning. In the ISO25000 series, "internal" and "external" refer to the fact that an attribute is relevant "internally" or "externally" with respect to the software development process. Thus, some attributes (e.g., reliability) are "external" for both the scientific literature terminology and the ISO25000 series. Others are classified differently, e.g., maintainability is "external" for the scientific literature terminology, but "internal" for the ISO25000 series.

3.3 Relationships between Internal and External Attributes

A direct consequence of the nature of external software attributes is that their quantification can be carried out by means of models whose independent variables are internal measures of the product, measures for the characteristics of its users, and measures of the interactions between the users and the product. These models must be probabilistic because many random variables can influence the

final result. When dealing with reliability, for instance, a failure may or may not occur in different sessions, given the software product used, its users, and a specified policy of interaction between product and users. In addition, using probabilistic models is also the theoretically sound way [22] of *measuring* external software attributes according to Measurement Theory [16, 27].

Models have been built for several external attributes. Like with internal software attributes, an external software attribute can be quantified by several different measures, i.e., several different models may exist for it. For instance, reliability can be quantified via the Jelinski-Moranda, the Musa-Okumoto, the Musa, the Littlewood-Verrall, and many other reliability models [21]. Fault-proneness, i.e., the probability that a software module contains a fault, has been quantified via several models, which have been used to classify modules as either faulty or non-faulty, based on a threshold on the value of fault-proneness itself [9, 14, 15, 28, 29] or a threshold on the slope of the fault-proneness model [23]. Reusability [7], maintainability [8], availability, accessibility, successability [30] can be quantified by means of models based on internal measures.

Note that it is the existence of *models* for quantifying external attributes that makes internal measures useful and justifies their collection from the field. As a matter of fact, when using an internal measure, one has always in mind the relationship of that measure with some tangible, practically useful external attribute, relevant to some software stakeholder.

As already stated, internal attributes *per se* are not relevant to software stakeholders. For instance, the sheer knowledge of *WMC* (Weighted Methods per Class) and *LOC* of a software product is not useful, unless there exists a known relationship between *WMC*, *LOC* and the measure of a software attribute of interest. If, for instance, *WMC* and *LOC* negatively influence the reliability of the software product, i.e., the probability that a software product does not experience a failure under specified conditions of use and in a specified time interval, knowing *WMC* and *LOC* can be useful to predict the level of reliability of the software product.

4 TECHNICAL DEBT AS AN EXTERNAL SOFTWARE ATTRIBUTE

As discussed in the Introduction, technical debt was proposed as a "metaphor" that helps software managers make decisions during the development process. So, effective decision-making requires that software managers quantify the amount of TD associated with software artifacts. In this section, we show how to quantify TD in a sound way, according to the basic concepts of Measurement Theory, which is the general reference framework for the definition of any measure [16, 27], and has been extensively used in Empirical Software Engineering [10].

In Measurement Theory, a measure for an attribute (e.g., size) of an entity (e.g., a software module) associates a value (e.g., an integer number) with the entity. Likewise, we can measure the TD (which is an attribute) of a software artifact (which is an entity) by associating a real number with the artifact, which quantifies the technical debt of the artifact.

The first consequence of this is that TD is not an artifact (or a process), as some authors have written by saying that some piece of code is TD. In Measurement Theory terms, this would equate to saying that a software module (or part thereof) is size. In financial debt terms, this would equate to saying that a house bought via a loan is financial debt, when instead the debt is the amount of money that must be paid to the financial institution that owns the loan. The implied meaning of saying that a piece of code *is* TD is that code comes with a burden, i.e., it *has* TD. Thus, code is an entity, associated with TD, which is therefore a software attribute. Following other authors [4], we use the term "Technical Debt Item" (TDI) to denote the entity a TD is related to.

Second, various TD types introduced by previous literature [4, 20] (e.g., requirements TD, design TD, code TD) are not different kinds of TD. These TD types actually denote the categories of entities (i.e., artifacts) of which TD is an attribute.

As discussed in Section 3, two categories of attributes exist: internal and external ones, depending on whether the attribute depends exclusively on the entity it is related to. We now show that TD is an external software attribute. Note that, in our discussion below, our concept of TD is broader than just "cost," i.e., financial debt, even though the term "debt" is naturally associated with financial matters. TD has been introduced and used in Software Engineering as a metaphor and as guidance for maintenance and should not be necessarily interpreted only as cost. Technical debts of different types exist. For instance, we may as well have time-related TD: the *time* for repairing quality issues in software systems to achieve an ideal quality level. Similarly, we could think of a technology TD: to optimize the maintainability of a piece of code, it is often the case that a technology gap has to be filled. For instance, if you have a piece of code written in Visual Basic, it is likely that it would become more easily maintainable if it was rewritten in Java.

So, we now show that, no matter the specific type of TD considered (be it cost, time, or any other industrially relevant concept), TD is an external attribute.

The heart of the matter is that the amount of TD related to a TDI depends on a number of factors, in addition to the TDI itself. Suppose that the TD associated with a TDI is quantified by a software manager when the TD is created. This quantification relies on a number of assumptions. For instance, the software manager assumes that he or she knows the process that will be used during debt repayment. However, the repayment process may very well change between the time in which the TD of a TDI is quantified and the time in which it is repaid. The same may very well be true for the technology used or the personnel involved in the repayment of the TD, which may change over time, even though the TDI remains the same. Even the requirements with respect to which repayment maintenance is carried out may change over time. So, TD is quantified based on the knowledge of a number of different factors, many of which depend on the human-intensive nature of software development.

From a modeling point of view, this means that the value of TD depends on a number of random variables. As a consequence, the quantification of the TD of a TDI is an *estimate* of it and not the actual value, which, theoretically, is unknown until the debt is perfectly repaid. As noted in [4], TD may not come due at all, so the project manager in the end may know the estimated value but not the actual value of TD. Note that, in general, TD may increase or decrease during the lifetime of the TDI. For instance, the introduction of a new technology may reduce the TD related to

a TDI, even though the TDI itself is unchanged. In addition, it will be often impossible to know in practice whether the debt has been repaid in full by refactoring, so even the TD value collected from the field is an estimate of the true value, instead of the true value itself.

The independent variables of the TD estimation model are used to estimate the TD, but they also help developers and maintainers figure out how a TDI should be changed, to reduce or eliminate its TD. For instance, if the estimated value of TD of a class is too high because it has an excessive value of *LOC*, then the class may be a God Class [26], which needs to be restructured in a specific way. If, instead, the estimated value of TD of the class is too high not because of *LOC*, which is in the normal range, but because *Ca* and *Ce* are too high, then we need to refactor the class and the software system in a different way, to reduce the coupling of the class.

Summarizing, a measure for TD is actually an estimation model, whose independent variables are factors that influence TD. These factors are the pieces of information that are used by project managers to quantify TD. The quality characteristics of a TDI, the development and repayment process, the technology used, the personnel, etc. can all influence TD. All of them are internal attributes of their respective entities. For instance, size, complexity, cohesion, and coupling are internal attributes of a software system. The number of the people involved in the development and repayment processes and their experience are internal attributes of the personnel. In conclusion, TD is an external software attribute.

Thus, TD is estimated by a function $TD(\underline{X})$, where $\underline{X}$ is the array of measures of the internal attributes that influence TD and therefore play the role of independent variables in the model. In our case, TD is an external attribute in which the "users" are actually the developers of the software system. Thus, we can rewrite the function as $TD(\underline{X}_c, \underline{X}_p, \underline{X}_t, \ldots)$, where $\underline{X}_c$ is an array of internal measures of code, $\underline{X}_p$ is an array of internal measures of the maintenance process, $\underline{X}_t$ is an array of internal measures of the technological platform. In what follows, we assume that $\underline{X}$ includes the measures of internal attributes of all the entities that affect TD.

Like all models that deal with randomness, a specific $TD(\underline{X})$ model is built by means of statistical or machine-learning techniques based on data from the field. The accuracy of $TD(\underline{X})$ also needs to be evaluated, to assess its practical usefulness. Clearly, a specific TD model refers to a specific type of TD, e.g., cost, time, etc., and the data collected from the field quantify that specific type of TD. This should not be surprising since, for instance, one may be interested in assessing TD in terms of additional cost, or additional person-months, or additional time, so different models will be built based on data on additional cost, or person-months, or time. Note also that several different models may exist even for the same type of TD. For instance, several models for the additional cost may very well exist, just like many models for cost estimation exist in the literature and in practical use.

5 THE ROLE OF TD PRINCIPAL AND INTEREST

The fact that the TD of a TDI depends on a number of factors partially mirrors what happens with financial debt. The value of the principal of financial debt is known and fixed at the time in which debt was created. However, the interest may change over time, for instance in the case of variable interest rates. Because of this variability, based on different scenarios that may occur, simulations of the amount of the debt over time are computed by financial institutions and provided to the customers. Also, financial debt may be renegotiated during its lifetime (e.g., by refinancing a loan), or even partially or totally canceled (e.g., in the case of bankruptcy).

As for TD, the principal itself may change over time. Suppose that a TDI of the current version of a software system is associated with some TD. Suppose also that no maintenance has been carried out on the system since the current version was released. The introduction of a new technology or simply of a new tool may reduce the initially estimated TD, i.e., the principal. Conversely, the sudden unavailability of adequate maintainers may cause an increase in the initially estimated TD. Like with financial debt, TD may never be repaid, if no further changes are carried out on a software system with TD.

As for interest, we need to identify the mechanisms through which interest accrues over time. The interest due to the TD related to a TDI needs to be calculated as a consequence of the TD alone. For instance, suppose that a variable in a class is defined as public, instead of being private or protected, as required by good Software Engineering practices. Suppose that the decision of leaving the variable public is a temporary expediency to make access to the variable easier for the objects of another specified class. Assume also that, in the development of the software system, no other classes are built that access the public variable. If this is the case— all other things being equal— no interest accrues, since the same refactoring can be carried out to repay the debt when the TD is introduced as at any later point during the development. If, instead, other classes are later built that also access the public variable, then there is additional TD to be repaid specifically due to the existence of the initial TD. This additional TD increases over time, i.e., there is interest that is due to the initial TD related to a TDI.

Therefore, just because time lapses does not mean that interest is added to the principal. This is a noticeable difference with respect to financial debt and interest.

Given the dependence of both the principal and the interest of TD on time, it is necessary that models of TD take time into account. The most natural way of doing it is probably by means of system dynamics models [2, 12]. In fact, we built a system dynamic model that accounts for the effects of TD in processes organized in sequences of sprints (e.g., according to the Scrum methodology) [17]. By simulating the model, we can show how both the TD principal and interest vary in time, depending on the policy adopted to repay TD.

6 TD RELATED TO DIFFERENT EXTERNAL ATTRIBUTES

A TDI is refactored to improve the internal quality of the software modules involved in the TDI, i.e., a set of internal software attributes. In turn, an improvement in internal software attributes is believed to induce an improvement in external software attributes. As we argued in Section 3, internal software attributes are practically useful only if they are related to external software attributes. In

other terms, refactoring a TDI with the only purpose of increasing the level of internal software attributes does not make industrial sense. When refactoring a TDI, one has always in mind, consciously or unconsciously, the improvement of some external attributes. For instance, one may rewrite a piece of code simply to make it compliant with some coding standard. This refactoring would not make practical sense if it did not improve the readability of the code, its modifiability, its maintainability etc., i.e., a few external software attributes. Note that refactoring a TDI may be done just to improve changeability. This means that the real advantage associated with refactoring will be achieved in the future, when the TDI is actually changed. So, it is still true that refactoring is performed to improve external software attributes, but the improvement is located in the future. Of course, in these cases refactoring is performed when it is expected that changes will be required in the future.

Repaying the TD related to a TDI is therefore ultimately related to one or more external software attribute. Thus, we argue that there exist several "facets" of TD. If, for instance, an inefficient algorithm was implemented in the initial version of a software module because of time or resource constraints, this software module is a TDI burdened with some *performance* TD. As another example, suppose that, initially, a quick-and-dirty GUI is built for a software system. This GUI is a TDI burdened with some *usability* TD.

For clarity's sake, note that the existence of TD facets is orthogonal to the existence of TD types. For instance, a TDI may have a TD quantified as cost (as TD type) that is related to performance (as TD facet), another TD quantified as time (as TD type) and also related to performance (as TD facet), and a TD quantified as cost (as TD type) and related to usability (as TD facet).

Categorizing TD according to the specific external software attributes it is related to can help project managers identify the weak points of their applications. Different external software attributes will have different degrees of importance in different software systems. For instance, performance is much more important than usability in an embedded application, while the converse may be true for a web-based system, whose success may primarily depend on the quality of the user experience.

Researchers can build external attribute-specific TD models that are more focused and accurate than a comprehensive TD estimation model that takes into account all external software attributes related to TD. We now show how these TD models can be built based on estimation models that quantify external software attributes. For illustration purposes, we use reliability as the external attribute of interest, i.e., the TD facet we are interested in. Reliability is an external attribute defined as the probability that a specified software does not have any malfunctioning in a specified time interval. Thus, reliability is quantified by means of a reliability model, i.e., a probability estimation model, whose independent variables quantify characteristics of the software, its users, and the way the users interact with the software.

Suppose that we have a reliability model $Rel(\underline{X})$, where $\underline{X}$ represents the array of its independent variables, which includes the measures of internal attributes of all the entities that affect reliability. Reliability is a quality that is important to the final users of a software product. Therefore, we can rewrite $Rel(\underline{X})$ as $Rel(\underline{X}_c, \underline{X}_u, \underline{X}_i)$, where $\underline{X}_c$ is an array of internal measures of code, $\underline{X}_u$ is an array of internal measures of the users, and $\underline{X}_i$ is an array of measures of the way the users and the software product interact. Note that, since TD and reliability have two different kinds of "users," $TD(\underline{X})$ and $Rel(\underline{X})$ do not have the same independent variables—however, they share the variables in $\underline{X}_c$, at least partially.

Take the current version SP_1 of a software product SP and suppose that $\underline{x}_{SP_1}$ is the array of values of the variables in $\underline{X}$ for model $Rel(\underline{X})$. For simplicity, assume for the time being that only one internal measure of SP appears in $\underline{X}$, say, $avgCa$, i.e., the average value of Ca of its modules. Suppose that $Rel(\underline{x}_{SP_1}) = 0.6$ and also that SP's requirements state that its required reliability level is 0.8. Then, SP has a reliability gap to bridge, i.e., SP's reliability must be increased by 0.2 for it to reach the required reliability level. The existence of this reliability gap implies the existence of one or more TDIs in SP_1.

To this end, we need to refactor SP to transform it from its current version SP_1 with value $avgCa_1$ for $avgCa$ to a new version SP_2 with a new value $avgCa_2$ for $avgCa$ so that $Rel(\underline{x}_{SP_2}) = 0.8$. Therefore, the reliability gap translates into a technical gap, which can be used to compute the technical debt due to the reliability gap. This reliability TD is a function $TD_{Rel}(\underline{X}_{SP_1}, \underline{X}_{SP_2})$ that depends on the values of $\underline{X}$ in SP_1 and in SP_2. The values of $\underline{X}_{SP_1}$ are a given, while the values of $\underline{X}_{SP_2}$ are computed based on the reliability model $Rel(\underline{X})$ and the desired reliability value 0.8. Note that knowing the value of $avgCa_2$ provides the developers with an indication that they can use when deciding about how to refactor SP_1.

Suppose now that two or more internal measures of SP appear in $\underline{X}$, say, $avgCa$ and $avgCC$ (the average cyclomatic complexity of the methods). We need to find two values $avgCa_2$ and $avgCC_2$ such that $Rel(\underline{x}_{SP_2}) = 0.8$. In general, there exist a set of possible pairs $< avgCa_2, avgCC_2 >$ such that $Rel(\underline{x}_{SP_2}) = 0.8$. Developers therefore have some more degrees of freedom when deciding how to refactor SP than in the previous case in which only one variable in $\underline{X}$ was an internal measure of SP.

$Rel(\underline{X}_c, \underline{X}_u, \underline{X}_i)$ clearly does not depend uniquely on $\underline{X}_c$, so, theoretically, one may think that the desired reliability level can also be achieved by changing the values of $\underline{X}_u$ and/or $\underline{X}_i$. However, this is impossible from a practical point of view. For instance, changing $\underline{X}_u$ means that we change the set of final users and changing $\underline{X}_i$ means that we change the way the users interact with SP. It is clearly not possible for the developers to change either one, so $\underline{X}_u$ and $\underline{X}_i$ must be taken as given. The only thing that developers can control is SP itself, so they can change the values of $\underline{X}_c$ only.

Note also that we used a *desired* value for reliability (i.e., 0.8 in our example) instead of the *ideal* reliability value, which is 1. Thus, the gaps in the level of some external attribute of interest are computed with reference to *desired* value and, in turn, the technical gaps depend on these *desired* values. We do so for a few reasons.

First, it is usually impossible to achieve the perfect level of an external attribute. For instance, the ideal level of reliability is 1 and the ideal level of performance in terms of execution speed is infinity. Thus, no matter what we do, there will always be a TD with respect to these perfect values.

Second, if we choose an "ideal" value for an external software attribute—even less demanding than the perfect one—, which we can use for all applications, we make an arbitrary selection, which does not take into account the specific needs and goals of a software

project. For instance, the ideal performance for an application may be much higher than the ideal performance for another type of application.

Third, it is unlikely that all of the qualities related to a project can reach their "ideal" values at the same time. Trade-offs must be made between conflicting qualities, so a project manager must set desired levels for qualities that can be achieved by the project based on the project's goals and needs. Note that these levels may change over time, which contributes to the intrinsic randomness of the nature of TD.

Trade-offs can be analyzed by using models for different qualities. For instance, suppose that, in addition to $Rel(\underline{X}_c, \underline{X}_u, \underline{X}_i)$, we also have a performance model $Perf(\underline{X}_c, \underline{X}_m, \underline{X}_{cm})$, where $\underline{X}_c$ is an array of internal measures of code, $\underline{X}_m$ is an array of internal measures of the machine or machines on which the software runs (which plays the role of the "user" of the software), and $\underline{X}_{cm}$ is an array of measures of the way the machine and the software interact. Again, $TD(\underline{X})$, $Rel(\underline{X})$, and $Perf(\underline{X})$ do not have the same independent variables, but they share the variables in $\underline{X}_c$, at least partially. Now take a variable X in $\underline{X}_c$. Refactoring the code to change X's value from x_{SP_1} to x_{SP_2} may have effects on both reliability and performance. These effects may be concordant or discordant. If they are concordant, there is no real trade-off to be analyzed. When the value of X is modified via refactoring, then both reliability and performance increase and reliability TD and performance TD decrease. If, instead, they are discordant, then models $Rel(\underline{X})$ and $Perf(\underline{X})$ can help us find a value of X such that both reliability and performance satisfy their respective requirements, if possible, and, if this is not the case, evaluate the effect on reliability TD and performance TD of possible refactoring.

Note that the fact that changes in the same variable may produce effects on more than one external attribute implies that the total technical debt $TD(\underline{X})$ is not necessarily the sheer sum of the TDs related to its facets. For instance, if X has concordant effect on both reliability and performance, then the same refactoring induces reductions in both reliability TD and performance TD, i.e., a double reduction, while the effect on the total TD is lower than the sum of the two reductions.

On a final note, the existence of desired values for external attributes TD implies the existence of thresholds for the independent variables. Once a desired value for an external attribute ea is set and a model $ea(X)$ for it is available based on an independent variable X, setting a desired level h for ea means having $ea(X) \geq h$. Thus, the set of values of X is partitioned in two subsets, i.e., the subset in which $ea(X) \geq h$ and the subset in which $ea(X) < h$. The values for which $ea(X) = h$ are the thresholds for X. If X is on the "wrong" side of a threshold, then there is TD related to ea. Note that this is the way bad smells are defined too. A bad smell occurs when some rule is violated. Thus, identifying a code TD basically implies identifying at least one bad smell. Setting project-specific desirable levels implies defining project-specific bad smells, instead of necessarily using general ones. The main difference between bad smells and TD is that bad smells are defined without taking into account the criticality/cost of the violation, while he notion of TD takes criticality/cost into account too.

7 TD REPAYMENT AND MAINTENANCE

In this section, we illustrate a few considerations concerning TD repayment and software maintenance. These considerations stem from typical situations and needs that arise in software development. We aim at showing that realistic issues concerning TD repayment and software maintenance are coherent with the view of TD as an external attribute described in the previous sections.

7.1 TD Repayment and Maintenance Issues

In Section 6, we showed how to introduce TD models based on the external attributes of a software product. However, there are two kinds of external attributes. Some are of interest of the final users, e.g., reliability, usability, and performance: we call these external attributes user-oriented attributes. Others are of interest of the developers, e.g., maintainability, readability, and changeability: we call these external attributes developer-oriented attributes. Suppose now that the software under consideration does satisfy its current requirements on user-oriented attributes, but it still has some inadequacies. Let us consider a couple of typical situations in code that determine the presence of these inadequacies and therefore of TD.

Example 1: our software contains duplicated code, that is, there are two or more identical (or very similar) pieces of code that perform the same (or very similar) functions.

Example 2: the code is poorly structured, so that a functional responsibility which is conceptually unitary has been spread over a number of classes or methods or functions.

In either case, we can essentially take two courses of action to address these inadequacies.

(1) We improve the code by removing the problems (duplicated code, bad structure, etc.) upfront, that is, we repay TD before its existence may generate interest. Then, when a change is requested concerning user-oriented attributes, we modify the new, improved code.
(2) Since the user-oriented qualities of our software are satisfactory, we do nothing until a change is requested concerning some user-oriented attribute. When this happens, we modify the code, which is still affected by the mentioned problems.

In case (1) above, refactoring to repay the TD may involve only marginal consequences in terms of user-oriented attributes. Actually, the final users may even experience a reduction in user-oriented attributes. For instance, suppose that the code contains several instances of the same duplicated code. We eliminate these instances by writing a method that does what the single instances of duplicated code do and by replacing each duplication instance with a call to this method. We improve developer-oriented qualities like modifiability, maintainability, testability, etc. that are of interest of the developers. At the same time, the performance experienced by the final users may be lower, because a method call may imply some delays, if compared to the direct execution of its statements.

So, we may want to pay off or reduce TD as an investment, because we believe that, if we wait to act (like in case (2)) until we are required to improve some user-oriented attribute, we will incur higher costs, time, etc. Partial or total TD repayment is an investment also because there is a substantial degree of uncertainty about its return. As we already mentioned, some TD may not come

due at all, so any refactoring investment carried out for this kind of TD is lost.

Concerning the cost of changes, in case (1), the overall maintenance cost is given by the cost of repaying TD plus the cost of changing the improved code; in case (2), the overall maintenance cost is given by the cost of changing the original code. The idea underlying acting like in case (1) is that changing the improved code costs less than changing the original problematic code. For instance, a change involving the duplicated code has to be replicated for all the duplicated pieces of code, while the equivalent change on the improved code would require modifying only one piece of code.

To make an informed decision, we need to evaluate the total costs for either option, as follows.

- Estimate the cost of TD repayment. This can be done by using the TD models developed.
- Identify the changes that will be likely requested in the considered period.
- Estimate the cost of changes performed on the improved code and on the as-is code. TD models allow for an estimate of these costs.
- Estimate additional effects: changes performed on improved code are likely faster; on the other hand, not repaying the TD leaves resources free to be employed in other (maybe immediately remunerative) tasks. Again, TD models help assess the extent of these additional effects.

In the cases discussed above, with the given code, the user-oriented attributes are satisfactory. Thus, all other things being equal, no action is actually immediately necessary and the software project manager could decide on whether or not to proceed with the removal of the inadequacies.

Of course, this is not always the case. An internal issue that causes TD may be associated with a user-oriented attribute that must be improved. For instance, the possibility of buffer overflow in an input operation has been discovered: the situation makes security breaches possible. In this case, the project manager has no choice but to proceed and correct the issue.

There is a whole spectrum of possibilities between optional and mandatory maintenance. Each of them has a different probability of occurring and a different associated TD. The procedure for evaluating whether or not to proceed to remove the TD outlined above for cases (1) and (2) can still be used, by taking into account the probability of occurrence of the need for maintenance.

Now, if any external software attribute is changed via software maintenance, the same applies to TD, which is an external software attribute, specifically a developer-oriented one. Several categories of maintenance have been identified, e.g., corrective, adaptive, perfective, etc. Thus, these categories also apply to TD. Each TD can be viewed as a combination of corrective TD, adaptive TD, perfective TD, etc. More specifically, each aspect of a TD can be viewed as such a combination. For instance, we may have a TDI with a usability TD, which needs to be repaid because the GUI has some flaws (corrective TD), needs to be changed to respond to chances in the windowing environment (adaptive TD), and needs to be improved because the current GUI version is minimal (perfective TD). Different categories of TD may be associated with different values of effort needed to repay TD.

7.2 An Example

A software development company develops a program that—among other things—computes the salary due to employees. The software project manager (PM) has to deal with the recurrent need for changing the program, since salary computation is subject to rules that are frequently changed. Therefore, salary computation has to be frequently adapted to new laws, agreements with labor unions, corporate rules, etc.

Currently, salary computation is performed in a class that contains employee's data (not only those related to salary computation), rules and parameters for salary computation, the actual code for computing the salary of different types of employees (the salary of managers is computed according to different criteria, with respect to other employees), and methods that performs other functions, not related to salary. In practice, class Employee is a "god class," according to the classification proposed by Riel [26]. Figure 1 shows class Employee in a UML class diagram. Let us call this diagram Model A.

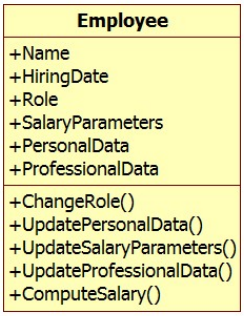

Figure 1: UML model of Employee god class.

Within the `Employee` class, a large and complex method computes the salary for all types of employees. Although salary is computed differently for employees and managers, part of the computation is similar: in the current software, the shared computation is performed by code that is duplicated in the employee salary computation and manager salary computation.

In conclusion, the current code is far from ideal: it contains a god class (`Employee`), a god method (`ComputeSalary`) and duplicated code. These "code smells" are easily identified by static analysis tools like Jdeodorant, PMD and many others.

So, the PM faces the situation described in Section 7.1; that is, he or she has to decide whether to repay the TD by refactoring the code and eliminating code smells, or live with the current situation until the code has to be changed because salary computation rules have changed. To decide, the PM can act as follows.

Given the output of the static analysis tools, the PM sketches a new situation, where the mentioned code problems are removed. Hence, he or she conceives a new software design, which is functionally equivalent to the previous one, but improves the internal

quality of the software. Figure 2 shows the new design in a UML class diagram. Let us call this diagram Model B.

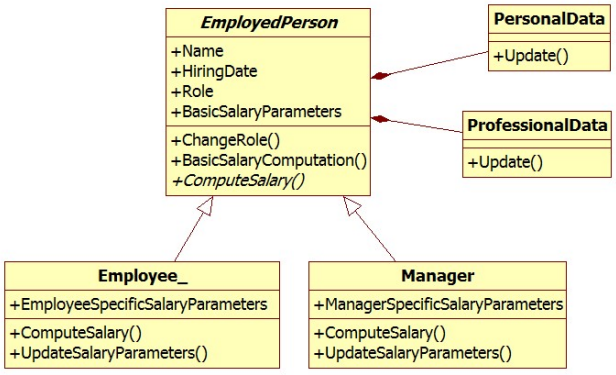

Figure 2: UML model of the refactored program fragment.

The difference between model A and model B can be described in terms of the refactoring activities listed in [13]. For instance, the "Move method" refactoring activity described by Fowler et al. in [13] is applied several times, to move methods originally belonging to `Employee` (like `UpdatePersonalData`, `UpdateProfessionalData` and `ComputeSalary`) to other classes.

Given the list of refactoring activities, the PM can easily estimate the time and effort needed to modify the code and achieve the situation described by model B (Figure 2). These estimates can be based on the PM's experience with refactoring, or can be derived using statistical models. In any case, let us call `eff_AB` the estimate of the effort needed to convert the current code into smell-free code, and `time_AB` the needed time.

The problem for the PM is now whether to implement the changes, i.e., repay the TD, knowing that doing so will require the estimated amounts of effort (`eff_AB`) and time (`time_AB`).

Since changes in salary computation rules happen quite frequently, the software company was able to collect measures concerning this type of changes. The analysis of the collected measures led to the creation of a few models, including:
`time(req_change, WMC, DIT, LOC, RFC, LCOM, CBO)` and
`effort(req_change, WMC, DIT, LOC, RFC, LCOM, CBO)`
where `req_change` provides indications concerning the type of required internal changes (e.g., 'parameter value change', 'parameter change', 'computation rule change', 'salary components change', etc.), and the code measures are those suggested by Chidamber et al. [5].

Based on his or her experience and on inputs from experts, the PM identifies a set of changes that will be likely requested in the short-medium term. Now, the PM can estimate the effort of implementing the changes starting from the situation described by model A. To this end, the PM uses the models with the measures of model A. Let `eff_AR` be the obtained estimate. Similarly, the PM can estimate the effort of implementing the changes starting from the situation described by model B. To this end, the PM uses the models with the measures of model B. Let `eff_BR` be the estimate.

Our PM has now a first piece of information: repaying the TD now will make the total cost of the changes `eff_AB+eff_BR`, while not changing the code until the changes are actually needed will cost `eff_AR`.

If `eff_AB+eff_BR` < `eff_AR`, repaying the TD now is clearly economically sensible. However, even if this condition is not satisfied, repaying the TD now could be economically sensible. Consider the case in which a new rule requires that salary computations be updated in two months. Our organization has to update its software in two months, otherwise it will incur a huge penalty. In such a case, even though `eff_AB+eff_BR` > `eff_AR`, it is likely that `time_BR` << `time_AR`, and penalty(`time_BR`)=0, while penalty(`time_AR`) is a huge amount. Therefore, repaying the TD now makes sense economically, in that it decreases the risk of losing money because the required changes will be delayed by the poor state of the code.

In the example above, it is quite clear that the notion of TD being used is that of an external attribute, since user requirements (salary computation rules) are involved, different types of TD are involved (concerning change effort and change time), different facets are possibly involved (although here we dealt only with functional adequacy). Finally, we note that models of external properties as functions of internal measures similar to those proposed in the literature have been used.

8 CONCLUSIONS

In this paper, we introduced a proposal for defining TD in a way that is is consistent with Measurement Theory, which is the general, authoritative reference for measurement and quantification activities. In this framework, TD appears to be a naturally external software attribute, because its value depends on the software under consideration and a number of other factors. The framework is general and allows one to define TD as cost, time, technology gap and any other industrially relevant quantities. We introduced the notion of facets of TD, whose quantification can help developers refactor software to eliminate its TD.

Future work will concern:

- the definition of models for the entire TD and quality-specific TD of an application;
- the practical use of TD models in development and maintenance: pilot studies will be first executed in controlled environment (e.g., empirical studies with students) and then the models will be applied in development and maintenance environments.

ACKNOWLEDGMENTS

This work was partly supported by the "Fondo di ricerca d'Ateneo" funded by the Università degli Studi dell'Insubria.

REFERENCES

[1] 2017. *Automated Technical Debt Measure – beta.* specification ptc/2017-09-08. OMG.
[2] Tarek Abdel-Hamid and Stuart E Madnick. 1991. *Software project dynamics: an integrated approach.* Prentice-Hall, Inc.
[3] 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. *Inf. Softw. Technol.* 70, C (Feb. 2016), 100–121. https://doi.org/10.1016/j.infsof.2015.10.008

[4] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn B. Seaman. 2016. Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports* 6, 4 (2016), 110–138. https://doi.org/10.4230/DagRep.6.4.110
[5] Shyam R Chidamber, David P Darcy, and Chris F Kemerer. 1998. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on software Engineering* 24, 8 (1998), 629–639.
[6] Ward Cunningham. 1993. The WyCash portfolio management system. *OOPS Messenger* 4, 2 (1993), 29–30. https://doi.org/10.1145/157710.157715
[7] Jehad Al Dallal and Sandro Morasca. 2014. Predicting Object-Oriented Class Reuse-Proneness Using Internal Quality Attributes. *Empirical Software Engineering* 19, 4 (2014), 775–821. https://doi.org/10.1007/s10664-012-9239-3
[8] John W. Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. 1996. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering* 1, 2 (1996). https://doi.org/10.1007/BF00368701
[9] Giovanni Denaro, Sandro Morasca, and Mauro Pezzè. 2002. Deriving models of software fault-proneness. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*. ACM, 361–368.
[10] Norman E. Fenton and James M. Bieman. 2014. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. Taylor & Francis. https://books.google.es/books?id=lx_OBQAAQBAJ
[11] International Organization for Standardization (ISO). 2005. ISO/IEC 25000:2005, Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE). (2005).
[12] Jay Forrester. 1961. *Industrial Dynamics*. Cambridge: MIT Press.
[13] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
[14] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2012. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. on Software Eng.* 38, 6 (2012).
[15] Taghi M. Khoshgoftaar. 2002. Improving Usefulness of Software Quality Classification Models Based on Boolean Discriminant Functions. In *13th Int. Symposium on Software Reliability Engineering –ISSRE, 12-15 November, Annapolis, MD*. https://doi.org/10.1109ISSRE.2002.1173256
[16] David H. Krantz, R. Duncan Luce, Patrick Suppes, and Amos Tversky. 1971. *Foundations of Measurement*. Vol. 1. Academic Press, San Diego.
[17] Luigi Lavazza, Sandro Morasca, and Davide Tosi. 2018. A System Dynamics Model of Technical Debt in Time-boxed Development Processes. In *Submitted for publication*.
[18] Jean-Louis Letouzey. 2012. The SQALE method for evaluating technical debt. In *Proceedings of the Third International Workshop on Managing Technical Debt, MTD 2012, Zurich, Switzerland, June 5, 2012*. 31–36. https://doi.org/10.1109/MTD.2012.6225997
[19] 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
[20] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 101 (2015), 193–220. https://doi.org/10.1016/j.jss.2014.12.027
[21] Michael R. Lyu (Ed.). 1996. *Handbook of Software Reliability Engineering*. McGraw-Hill, Inc., Hightstown, NJ, USA.
[22] Sandro Morasca. 2009. A probability-based approach for measuring external attributes of software artifacts. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09, Lake Buena Vista, FL, USA, October 15-16, 2009)*. IEEE Computer Society, Washington, DC, USA, 44–55. https://doi.org/10.1109/ESEM.2009.5316048
[23] Sandro Morasca and Luigi Lavazza. 2017. Risk-averse slope-based thresholds: Definition and empirical evaluation. *Information & Software Technology* 89 (2017), 37–63. https://doi.org/10.1016/j.infsof.2017.03.005
[24] 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 2011, Waikiki, Honolulu, HI, USA, May 23, 2011*. 1–8. https://doi.org/10.1145/1985362.1985364
[25] Narayan Ramasubbu, Chris F. Kemerer, and C. Jason Woodard. 2015. Managing Technical Debt: Insights from Recent Empirical Evidence. *IEEE Software* 32, 2 (2015), 22–25. https://doi.org/10.1109/MS.2015.45
[26] Arthur J Riel. 1996. *Object-oriented design heuristics*. Addison-Wesley Publishing Company.
[27] Fred Roberts. 1979. *Measurement Theory with Applications to Decisionmaking, Utility, and the Social Sciences*. Encyclopedia of Mathematics and its Applications, Vol. 7. Addison-Wesley.
[28] Norman F. Schneidewind. 2001. Investigation of Logistic Regression as a Discriminant of Software Quality. In *7th IEEE Int. Software Metrics Symposium–METRICS*. https://doi.org/10.1109/METRIC.2001.915540
[29] Raed Shatnawi. 2010. A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems. *IEEE Trans. Software Eng.* 36, 2 (2010). https://doi.org/10.1109/TSE.2010.9
[30] Abbas Tahir, Sandro Morasca, and Davide Tosi. 2013. Towards Probabilistic Models to Predict Availability, Accessibility and Successability of Web Services. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2013, October 27 - November 1, 2013, Venice, Italy)*.