ISSN (e): **2250 – 3005** || *Volume, 07* || *Issue, 07*|| *July – 2017* ||
International Journal of Computational Engineering Research (IJCER)

Towards A Modularity-Based Technical Debt Prioritization Approach

[1]srinivas Mishra, [2]sai Krupa Nayak
Gandhi Institute of Excellent Technocrats, Bhubaneswar, India
Sanjay Memorial Institute of Technology, Berhampur, Odisha, India

ABSTRACT
Technical debt (TD) refers to aspects in software develop- ment that can have short-term benefits (e.g., faster time-to- market) but may be detrimental in the future (e.g., due to decreased software modifiability). TD management (TDM) deals with activities to control TD, deciding which debt to repay and which to defer, and monitor the effect of the in- curred TD on business goals and productivity. Effective TDM requires prioritization of TD items in order to re- pay the debt with the strongest negative effects on business priorities and software quality first. This paper gives an overview of existing approaches for TD prioritization and aims to analyze commonalities and differences in order to extract prioritization rules and metrics. We focus on archi- tecture, design and code debt that negatively impacts modu- larity and propose a prioritization approach for modularity- relatedTD.

Categories and Subject Descriptors
D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]:Management—costestimation,timeestima- tion,productivity

Keywords
technical debt, technical debt prioritization, design debt, modularity

INTRODUCTION

Since Ward Cunningham coined the term technical debt (TD) in 1992, its usefulness to manage software projects more proactively and communicate to non-technical stake- holders has been widely appreciated [3, 13]. The metaphor refers to shortcuts taken during development which speed up development time in the short-run but hamper produc- tivity and software quality in the long-run [12]. Typical rea- sonsforintentionallyincurredTDincludebusinessneeds,

Explicitly managing TD has several benefits: it is an effective way to manage software projects proactively [9], streamline refactoring activities [19], and monitor develop- ment progress [9]. Proactive management is supported by monitoring trends and acting upon them. For example, when a development team fails to write sufficient software tests, thereby incurring test debt, this problem can be identi- fied and addressed early [9, 10]. Streamlined refactoring de- cisions are supported by TD prioritization methods which facilitate identifying high-impact debt [19]. Lastly, active tracking of TD over time allows to evaluate if TD is handled successfullyorincurreduncontrollably,sothatcorrectiveac-

tion can be taken if needed[9].
Li et al. identified 8 TDM activities, one of which is TD prioritization. TD prioritization means ranking identified TD items according to a set of predefined rules, e.g., using a cost-benefit approach [12]. For example, in an object- oriented project, one TD item may be a class violating the SingleResponsibilityPrinciple.Anotheronemaybelacking documentation for an interface. The first one may take more timetofixbutprobablyalsoprovideabiggerpositiveimpact on the software quality – this depends on the organization's or the project's priorities[12].
Activities related to TDM which typically take place be- fore prioritization are TD identification to identify inten- tional and unintentional debt, and TD measurement to es- timate cost and benefit of TD items in one form or another. Further activities include TD monitoring to track changes in cost and benefit for TD items, TD repayment to eventually pay off selected TD items, and TD communication to raise awareness for TD among stakeholders [12].
TD prioritization is an essential part of the TDM process

because it can be implemented in a way that it reflectsnot only technical, but also business priorities, and these ulti- matelytakeprecedenceinpractice[3].Severalframeworks forTDprioritizationhavebeenintroducedbyresearchers[9, 6];however,Lietal.recentlyidentifiedseveralopenchal- lengesconcerningTDprioritization.Theseincludelacking toolsupport,waystoprioritizeTDtomaximizebenefit,and whichfactorstotakeintoaccountforprioritization.[12].
Several approaches for TD prioritization have been intro- duced,mostofwhichrelyonsourcecodeand/ordesignmet- ric measurements to prioritize TD [19, 9, 6]. All approaches use some way to measure the severity of a particular TD item (e.g., its interest) and the effort involved in fixing it (i.e. its principal). These values can then be used to pri- oritize the TD items. We will explain some approaches in more detaillater.
The modularity-based approach presented in this paper is mainly concerned with design debt reflected in source code. In this regard, it is complementary to the SQALE method. To our knowledge, there has been no previous research on TD related to modularity in particular. We address this gap and present a framework to prioritize this type of debt.
Thepaperisstructuredasfollows.Insection2,weprovide the necessary background and analyze previous approaches to identify commonalities, differences, and limitations. Sec- tion 3 presents the modularity metrics identified in this pa- per and provides a brief rationale for each. In section 4, we demonstrate a TD prioritization approach based on the introduced metrics. Section 5 concludes this research by mentioninglimitationsofourapproachandfuturework.

1. BACKGROUND

Forthepurposesofthispaper,wewillfocusonarchitec- tural, design, and code debt. These correspond to three of the ten major types of TD identified by Li et al. in their recent mapping study [12]. We also note that we consider intentionalandunintentionaldebtequallysincebothcould be equally detrimental to softwaremodularity.
TDM relies heavily on TD prioritization to streamline maintenance activities. Yet, of the 29 tools evaluated by Li et al., only the SQALE plugin for SonarQube and the Sonar TD plugin supported TD prioritization [12]. We will expand upon this area by addressing modularity as a sub- topic and hope to encourage similar research in other niche topics which can serve as a basis for a more comprehensive framework in thefuture.

Existing TD PrioritizationApproaches

Previous research has developed several TD prioritization approacheswhichexhibitmanycommonalitiesbutalsosome foundationaldifferences.

Finance-basedApproach

Guo and Seaman proposed a portfolio approach for TD prioritization [6]. This approach offers a new perspective on TDM by reusing established portfolio management strate- gies from the finance domain. Here, TD is considered an investment and TD items are considered assets of a portfo- lio. Portfolio management aims to reduce risk andmaximize return on investment (with TD being the investment). This maps nicely to the issue of TD prioritization. For this ap- proach, important characteristics of each TD item (such as principal,interestamount,interststandarddeviation,and

correlations between TD items) must be estimated and doc- umented. After that, a portfolio management model such as the Modern Portfolio Theory model is used to extract the best assets (i.e., TD items). The result is a subset of the assets which reduces risk and maximizes return. Thus, TD itemswhicharenotpartofthisportfolioshouldbefixedfirst [6]. Guo and Seaman note that there are several incompati- bilities between financial assets and TD items which needto be considered: In finance, assets are divisible, and their cost and interest are known in advance – both of which are not the case with TD. The approach suggests using quantita- tive methods for the estimations if possible, and qualitative methods otherwise. Also, the performance of this approach in practice was not tested[6].

Design Quality PrioritizationApproach

Anotherprioritizationapproachwhichfocusesondesign debt in particular was presented by Zazworkaet al. [19]. Morespecifically,theirpaperfocusesongodclasses. They presentacost-benefitapproachtodecidewhichgodclassto refactor first, based on three metrics: a) weighted method count;b)tightclasscohesion;andc)accesstoforeigndata. For each metric, an acceptable threshold is defined. Zaz- workaetal.notethatonecanassumethecostofrefactoring to increase with the distance of the class from the thresh- old. For instance, a class with 500 methods will typically takemoreefforttorefactorthanonewith50methods.This argumentismadeforeachofthethreemetrics.Changehis- tory data are used to estimate change likelihood as wellas defectlikelihoodforeachclass.Thedistanceofaclassfrom thethresholdsthendefinesthecostwhereasthechangeand defectlikelihoodsdefinethebenefitofrefactoringthegod class.Acost-benefitmatrixcanbeusedinordertogivean effectiveoverviewonwhichgodclassestoprioritize[19].

Metrics-basedApproach

The SQALE method is a metrics-based approach which addresses eight so-called characteristics which are divided into sub-characteristics and eventually source code require- ments. The used characteristics are ordered by the time they become important in the file lifecycle. They are, in order, testability, reliability, changeability, efficiency, secu- rity, maintainability, portability, and reusability [9]. Re- quirements represent the definition of "right code", e.g., no commented out code, no interfaces without documentation, or no public class attributes. Note that the modularity char-acteristic is related to changeability, maintainability and reusability. In fact, Li et al. categorized modularity as a sub-attribute of maintainability, adhering to ISO 25010 [12]. However, the SQALE approach does not explicitly address modularity and is restricted to only code-related debt [9, 10]. In order to estimate the amount of TD, SQALE re- quires a remediation function for each requirement. This represents the estimated time needed to fix a TD item (the principal), e.g., 30 minutes to add documentation to an in- terface. Similarly, each TD item must be associated with a non-remediation function – the estimated cost for not repay- ing the debt [10]. These are simply accumulated torepresent the TD for modules, subsystems or the whole system. The SQALEmethodalsocontainswaystovisualizethisTD,such as the Rating Grid, the SQALE Pyramid, and the Debt Map [9,10].

Analysis of ExistingApproaches

Wewillnowevaluatecommonalities,differences,andlim- itations of the aforementioned prioritizationapproaches.

Commonalities

Alltheapproaches**exhibitsomedefinitionofcostand benefit** which is used for prioritization. In the SQALE ap- proach, the remediation and non-remediation functions cor- respond to principal and interest, respectively. Thus they both represent costs. However, the non-remediation func- tions also represent the benefit achieved when repaying the debt. In the design debt prioritization approach introduced by Zazworkaet al., the distance of a class from the threshold can beseen as a cost (the principal for fixing the item) while change and defect likelihood act as a measure of benefit. In the portfolio approach, interest cost is explicitly defined as estimated interest amount and interest standard deviation as part of the TD items. Principal is also an explicit part of each TD item (equivalent to the remediation function in SQALE) [6]. Benefit is not documented explicitly but can be derived from the interestmeasures.

Another commonality between the SQALE method and the design debt prioritization method is that **both rely on metricsasabasisformeasuringandprioritizingTD**. InSQALE,therequirementsdefinedbytheorganizationrely

on metrics in order to evaluate adherence to these require- ments. For example, a requirement like"method should have nomorethan100LOC"wouldrelyonametricsuchas"lines ofcodeinmethod".SonarQubeismadetoprovidesuchmet- rics which allows the SQALE method to be implemented effectively in SonarQube. Similarly, the god class prioriti- zation is explicitly based on the metrics weighted method count, tight class cohesion, and access to foreign data to identify and measure the severity of each god class. Defect likelihood and change likelihood are two more metrics used toestimatetherelevanceandseverityofthedebt.
Additionally, the SQALE method and the god class prior- itization approach both include **visual representationsto aidthedecisionprocess**.Theportfolioapproachcould be extended accordingly; however, the approach has yet to be refined and tested in practice. Especially in the SQALE method, visualizations play an important role. Various visu- alizationshave been introduced, including the Rating Grid, the SQALE Pyramid, and the Debt Map [9, 10]. Accord- ingly, the SonarQube plugin implements suchvisualizations. One can assume that these support both understanding and decision-making.

Differences
The unique feature of the portfolio approach is that it reuses existing knowledge from the finance domain. This is potentially an important advantage because portfolio ap- proaches from finance are well-understood and established. Other approaches usually rely on previous research in fields such as software design, software quality measurement, and coding best practices. For instance, the design debt priori- tization approach relies heavily on previous research on god classes which is a well-known issue in software design and softwareevolution.
The SQALE method is unique in that it has already been widely adopted in the industry through its implementation in SonarQube. To our knowledge, no other tool for TDM isusedasmuchinpractice.Infact,thereisnoresearchon

the real-world implementation of the portfolio approach. Thepresentedcomparisonalsohighlightsthefactthat,
whilemostapproachesmakeuseofpredefinedmetrics,other approaches such as the portfolio approach are also conceiv- able. These may pose a good opportunity for research since previous research has focused on metrics-based methods. The approach presented in this paper will rely on metrics aswell.

Limitations
The design debt prioritization approach introduced by Za- zworkaet al. is obviously limited to god classes in the form presented in the paper [19]. However, the principles may be reused for similar design flaws. For example, a similar ap- proach for improper inheritance structures may use metrics suchas"numberofchildclasses","depthofinheritancetree", or "composition not preferred over inheritance". Another limitation of the approach as presented in the paper is the fact that it relies on historical data to estimate change and defect likelihood. Thus, the quality of the estimate depends on the amount of available history, making it less applicable for newerprojects.
With their portfolio approach, Guo and Seaman provide a new perspective on TDM. However, the implementation of this method in practice, its assumptions, conditions, and applicability remain to be evaluated when applied in the context of TD. They also mention some general guidelines based on finance which need further evaluation. For ex- ample, Guo and Seaman propose to prefer many small TD items over one big TD item – this diversification promises to decrease risk [6]. Similarly, one should prefer TD items with low positive correlations. Guo and Seaman proposed further studies for empirical evidence[6].
The SQALE method is limited to code-related debt which is only one of the ten types of TD [12]. However, other types of TD such as design debt or test debt are typically associated with the code debt. Thus, fixing code debt can also mitigate other types of debt. Also, we identified that the SQALE method lacks an explicit measurement of mod- ularity. The characteristics, sub-characteristics, and require- ments of the method provide an opportunity to tweak and/or extend it. Thus, the method could by extended by either adding modularity as a characteristic or as a sub-charac- teristic of maintainability, as in ISO 25010 [12]. The results of this paper can be used to add modularity requirements andareinsofarcomplementarytotheSQALEapproach.

2. MODULARITYMETRICS

The modularity of a software system refers to the capabil-ityofitsmodulesandsubsystemstofunctionasautonomous modules and provide their services outside the

original sys- tem [1]. The major benefit of such modularity is the option to substitute system components if a superior implementa- tion becomes available. Since this option is available but not obligatory, and potentially improves system design, it provides a positive net value[17].

Existing ModularityMetrics

Inordertomeasureandevaluatemodularity,variousre- searchers have gathered a catalog of modularity metrics. In 2007, Sant et al. presented 11 architectural metrics to measure modularity [15]. They are based on the princi- plesthatasingleconcernshouldtypicallyberealizedby a single component, that shared data and state between components should be minimized, and that the complex- ity of components should be reasonable. Additionally, Li and Henry gathered maintainability metrics, some of which can be applied for modularity concerns [11]. These include depth of inheritance tree (DIT), lack of cohesion of methods (LCOM), and number of child classes (NOC). A rationale for each is givenlater.

Another way to discover modularity requirements is by looking at research on modularity violations. Wong et al. presented the CLIO tool which detects such violations by comparing which components should change together and which did change together according to version control his- tory [18]. This indicates the concept that architectural and designmetricsshouldbemappedtomeasurablecodemetrics if possible in order to evaluate consistency with the archi- tecture anddesign.

Metrics for modularity can be mapped to source code re- quirements, equivalent to those introduced in the SQALE method. By identifying and prioritizing such metrics, we will present a prioritization approach for modularity-related TD which can be integrated into the SQALE method if de- sired.

A Catalog of ModularityMetrics

The modularity debt prioritization approach introduced in this paper, like many others, is based on metrics. We present a catalog of metrics in Table 1. Typically, archi- tectural metrics imply design metrics which in turn imply code metrics. Note that the derivation of metrics ends on thedesignlevelifitheyaremeasurableonthatlevelalready; however,mostofourmetricsmapdirectlytosourcecodere- quirements.Inordertodiscoverthemodularitymetrics,we relied on previous research on architectural bestpractices, modularityissues,andpreviouslypresentedmetrics.Most predominantly, we derived design metrics fromarchitectural practicesandthenderivedcorrespondingsourcecodemet- rics. In most cases, this yielded well-known metrics from previousresearch.

Note that there are two trends here. First, the derived metricsoftencorrespondtogooddesignorcodingpractices. Second, architectural metrics tend to imply severaldesign metrics which in turn tend to imply several code metrics. Thismakesthemethodologyveryfruitfultoderivearange of source codemetrics.

Rationale

Table 1 shows all metrics identified in this paper and how they are derived from each other. We will give a brief ratio- nale for each to explain in which ways they support modu- larity. References to other work are given where applicable.

Low Coupling BetweenModules

Low coupling between modules is a major architectural requirement for modularity and reusability [1]. Based on this principle, we derived several design requirementsand metrics. First, components should communicate via well- defined interfaces. Thus, on code level, developersshould always refer to the most general type possible [2] in order toabstractfromtheconcretesubtypeandimplementation. Next, associations and, more strongly, compositionsintro- ducedependenciesbetweencomponentsandshouldthere- forebeminimized.Oncodelevel,thiscanbemeasuredby the number of imported types. Also, the message passing coupling can be measured by the number of method calls on other classes [11]. Next, the use of intermediaries decouples components by adding a layer of abstraction for commu- nication. Such patterns include Facade, Mediator, Proxy, Strategy, Factory, Publish-Subscribe, and Blackboard [1]. Note that these are architectural and design-level patterns, yet could be measured semi-reliably on source code level byrelying on naming conventions or stereotypes. A strong formofcouplingisinheritance[11]whichimpliestwodesign- level metrics, the number of parent classes (i.e., the depth in the inheritance tree in single inheritance languages), and the number of child classes (NOC). Arguably, inheritance couples components stronger than association[2].

Proper Distribution ofFunctionality

Second, proper distribution of functionality is a well-known challenge for software architects [1]. Design-level require- ments derived from this include tight class cohesion (TCC) [14] and proper reuse of functionality. Tight class cohesion can be measured on code level by clusters of methods that share no common data at all (LCOM) [11]; the overall num- ber of lines of code (LOC) may also be indicative of the cohesion since very large classes tend to handle variouscon- cerns [19]. The proper reuse of functionality is reflected by the lines of duplicated code, corresponding to the don't- repeat-yourself principle of object-orienteddesign.

Information-HidingInterfaces

Third, we chose the use of information-hiding interfaces [17] as a separate architectural requirement since it yields severaldesignandcodingguidelines.Onemetricisthenum- ber of interfaces in comparison to the number of classes to estimate how widely information hiding is employed. This is related to "communication via interfaces" above. Another important metric is the number of public class attributes [4, 19] (excluding explicit class constants) since these violate encapsulation. More generally, we propose to measure the percentages of private, protected, and public attributes in a class (the naming convention is based on Java and similar languages). This allows to estimate the strength of encap- sulation in more detail. For instance, private attributes can reduce inheritance coupling because even subtypes cannot access privateattributes.

3. MODULARITY-BASEDTDPRIORITIZA- TIONAPPROACH

Overview

Based on the presented catalog of modularity metrics, dif- ferent prioritization strategies may be defined. In the fol- lowing, we present a cost-benefit approach which balances principal against interest amount and probability – similar to remediation and non-remediation functions respectively. We note that various other prioritization strategies may be defined based on these samemetrics.

In order to derive concrete TD items from the metrics, we must define a threshold based on our definition of "right code". For example, we may specify the threshold for the depth of a class in the inheritance tree to be lower than five. Then, in order to prioritize the TD items, we assign princi- pal, estimated interest amount (EIA), and estimatedinterest

Architecture level	**Design level**	**Code level**
Low coupling between modules	Communication via interfaces [2]	Number of type references replaceable by more general type [2]
	Number of associations to other classes (FAN OUT [4])	Number of imported types [19]
		Number of method calls on other classes[11]
	Numberofassociations from other classes to this class (FAN IN)[7]	Number of imports of this class in other classes
		Number of classes calling methods from this class
	Number of intermediaries	Number of usages of Facade, Media- tor, Proxy, Strategy, Factory, Publish- Subscribe, Blackboard and similar pat- terns [1]
	Number of parent classes / Inheritance depth (DIT) [11]	
	Number of child classes (NOC) [11]	
Proper distribution of func- tionality	Tight class cohesion (TCC) [19, 4]	Number of clusters of methods without a shared variable (LCOM) [11]

		Source lines of codes in class (LOC) [11, 4]
		Weighted method count (WMC) [11, 19]
	Proper reuse of functionality [19]	Lines of duplicated code (SEC) [8]
Components have information-hiding interfaces	Percentage of interfaces vs. classes	Number of classes
		Number of interfaces
	Number of public class attributes (excluding con-stants) [4, 19]	Number of private attributes
		Number of protected attributes
		Number of public attributes (NOPA) [4]

Table 1: A catalog of modularity requirements and metrics.

probability (EIP). This is not a trivial task because the esti- mations depend on organizational and technical factors [4], including developer skills, interdependencies between TD items, and projected future changes. Therefore, the estima- tions must be performed by each organization and project individually. Principal, EIA, and EIP have all been used in previous research [6, 16]. The principal defines the cost for fixing the TD item and thus corresponds to a remediation function in the SQALE approach. Similarly, interest amount and probability define the cost for not fixing the TD items and thus correspond a non-remediation function. The ratio- nale behind the probability is that modules which will likely never be changed in the future should have accordingly low priority [5].

Cost-BenefitFormula

Once we have assigned principal, EIA, and EIP, we can use these to prioritize the TD items. For the purposes of this paper, we will assign to each TD item I the priority P (I) calculatedas

EIA · EIP

whichrepresentsacost-benefitapproach–dividingbenefit by cost. Thus, the TD items I with the highest priority P (I) should be repaidfirst.

Procedure

To use this approach, several steps and guidelines should be followed. A prerequisite is the ability to measure code- related debt. Ideally, design debt can be measured as well. The procedure is as follows:

1. AssignathresholdtoeachmetrictoderiveTDitems.
2. Estimateprincipal,EIA,andEIPforeachTDitem.
3. Calculate P (I) for eachitem.

For step 1, Fontana et al. provide some guidance in a re- cent study [4]. However, thresholds must be defined and evaluated by the organization based on what works in their context. For step 2, you should take into account your devel- opment team's skills, organizational constraints, and histor- ical data to improve the estimations [16, 19]. Asmentioned

P (I) =

.

Principal

above,thisisacomplextaskduetomanyinfluencingfac-

tors.Sincethequalityoftheapproachdependsdirectlyon

Thisisaverysimplewaytoassignausefulpriorityscore
thequalityoftheestimations,werecommendmonitoring

TD Item	Metric	Threshold	Principal	EIA	EIP	P(I)	Rank
3 public attributes in SampleClass	NOPA	0	5h	3h	20%	0.12	2
4 clusters in Another-Class	LCOM	2	20h	10h	50%	0.25	1
Depth of SampleClassis 7	DIT	5	25h	5h	25%	0.05	3

Table 2: List of sample TD items with threshold, principal, and interest.

and adjusting the estimations. To identify the TD items and measure their related metrics, you may use an analysis tool such asSonarQube.

Applying theApproach

To illustrate the presented approach and make it more tangible, we present a brief example. We assume we have derived the TD items listed in Table 2.

The example is set up in a way that AnotherClassis changed much more frequently than SomeClass, thus the higher EIP. We can see that the debt related to Another-ClassshouldberepaidfirstbecauseP(I)ishigherthanthat of any other TD item. This originates from the fact that AnotherClassis changed often and thus has high interest. The next TD item to fix would be the first one because its principalislowcomparedtothatofthethirditem.

CONCLUSION & FUTUREWORK

In this paper, we have introduced a modularity-based TD prioritization approach based on several metrics we ex- tracted or derived from previousresearch.

In its current state, the presented approach lacks a way to systematically estimate and assign values such as princi- pal, estimated interest amount (EIA), and estimated interest probability (EIP) which is not a trivial issue in TDM. Like other current approaches, we rely on user input for these estimations [9, 6]. Tools to guide the estimation of such val- ues remain future work. Such tools should also consider the severity of violations, e.g., by measuring the distance from thedefinedthresholds[19]orotheruser-definedmetrics.We also note that we do not consider the presented metricsfixed and expect future work to extend or refine the catalog. In addition, future research focusing on niches other than mod- ularitymay provide further insights which can pave the way to a comprehensive TDM approach. Also, other priority cal- culationsmay be defined based on the provided metrics and resulting TD items. The implementation of the presented approach in practice remains to be evaluated to generate empiricaldata.

REFERENCES

[1] L. Bass. Software architecture in practice.PearsonEducation India, 2007.

[2] J.Bloch.EffectiveJava.JavaSeries.Pearson Education,2008.

[3] F. Buschmann. To pay or not to pay technicaldebt.Software, IEEE, 28(6):29–31, 2011.

[4] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda. Towards a prioritization of code debt: A code smell intensity index. In Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on, pages 16–24. IEEE,2015.

[5] G.Technicaldebt:Strategies&tacticsforavoiding& removing it. http://blogs.ripple-rock.com/SteveGarnett/2013/03/05/TechnicalDebtStrategiesTacticsForAvoidingRemovingIt.aspx. Accessed: 2015-12-01.

[6] Y. Guo and C. Seaman. A portfolio approach to technical debt management. In Proceedings of the 2nd Workshop on Managing Technical Debt, pages 31–34. ACM,2011.

[7] S. Henry and D. Kafura. Software structure metrics based on information flow. Software Engineering, IEEE Transactions on, SE-7(5):510–518, Sept1981.

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

[9] J.-L. Letouzey. The sqale method for evaluating technical debt. In Proceedings of the Third International Workshop on Managing Technical Debt, pages 31–36. IEEE Press,2012.

[10] J.-L. Letouzey and M. Ilkiewicz. Managing technical debt with the sqale method. IEEE Software, 29(6):44–51,2012.

[11] W. Li and S. Henry. Object-oriented metrics that predict maintainability. Journal of systems and software, 23(2):111–122,1993.

[12] Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. Journal of Systems and Software, 101:193–220,2015.

[13] E.Lim,N.Taksande,andC.Seaman.Abalancing act: what software practitioners have to say about technical debt.Software, IEEE, 29(6):22–27,2012.

[14] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In Software Maintenance, 2004.

Proceedings. 20th IEEE International Conference on, pages 350–359. IEEE, 2004.

[15] C. Sant Anna, E. Figueiredo, A. Garcia, and C. J. Lucena. On the modularity of software architectures: A concern-driven measurement framework. In SoftwareArchitecture,pages207–224.Springer,2007.

[16] C. Seaman and Y. Guo. Measuring and monitoring technical debt. Advances in Computers, 82:25–46, 2011.

[17] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. SIGSOFT Softw. Eng. Notes, 26(5):99–108, Sept.2001.

[18] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In Proceedings of the 33rd International Conference on Software Engineering, pages 411–420. ACM,2011.

[19] N. Zazworka, C. Seaman, and F. Shull. Prioritizing design debt investment opportunities. In Proceedings of the 2nd Workshop on Managing Technical Debt, pages 39–42. ACM,2011.