Measuring Architectural Technical Debt

MASTER'S THESIS

submitted to the Faculty of Science
of Radboud University in Nijmegen
for the degree of

MASTER OF SCIENCE

in

INFORMATION SCIENCE

by

Michail Kuznetcov
born in Moscow, Russian Federation 1984

Radboud
University
Nijmegen

ABN AMRO Bank N.V.
Gustav Mahlerlaan 10 (1082 PP)
Amsterdam, the Netherlands
www.abnamro.com

Radboud University Nijmegen
Comeniuslaan 4 (6525 HP)
Nijmegen, the Netherlands
www.ru.nl

Author: Michail Kuznetcov
Student number: s4340132
E-mail: michail.kuznetsov@gmail.com

Abstract

To develop the concept of technical debt in application to modernization of the architecture of the large information systems, such as the host company of this research, there is a need in modern measurement approach. Based on the state of the art described in the scientific literature, we have developed a model which aims to identify and quantify individual technical debt items.

The method itself includes a measurement model and accompanying estimation guidelines. The model's structure is based on several theoretical concepts in the research area of technical debt. During the research we specifically focused on architectural technical debt issues. We developed a taxonomy of types of debt which included for the architectural level the enterprise architecture methodology that was used by the company. Estimation guidelines were developed based on the professional experience of the participants to provide the reference for estimating each of the assessed types of debt.

Throughout the research we have collected data for several specifically chosen systems within the host company's IT landscape. Data was collected from the company's employees using a specifically designed collection tool. During data collection participants estimated the amounts of debt principal and interest associated with each system against proposed debt categories.

To perform validation of the developed model we have studied the perception of practitioners by using a feedback questionnaire. This helped us to assess feasibility of the method and according to the results, the company's practitioners find the model and the approach in general useful, understandable, and compatible with industry experience.

Keywords: technical debt, architectural debt, large information systems, TOGAF

University supervisor: Joost Visser, j.visser@sig.eu
Company supervisor: Martin Paris, martin.paris@nl.abnamro.com

Acknowledgments

This thesis would not have seen the light without the help and support of a number of people. My gratefulness for these people is simply inexpressible, below I would like to say personal thanks to most of them.

First and foremost, I would like to thank my company supervisors and mentors Martin Paris and Raghu Warrier for sharing their experience and time during practical part of the research. Their constant availability, support and guidance were always helping me to achieve expected results. Next person I would like to thank is Joost Visser from Radboud University Nijmegen, my academic mentor, for sharing analytical approaches towards the research, and always giving logical comments on my work.

Having an opportunity to do my thesis at such a large company with modern approaches towards IT as ABN AMRO was an amazing final step in my education in Netherlands. And I would like to say personal thanks to people who helped me on my way to start it – Ruth Koppenol, George Jansen and Jan Robat. Being a part of such a professional team had a great impact on my personal and professional skills.

Specifically I would like to thank ABN AMRO developers, testers and architects – Namrata Sen, Jaap Teeuwen, Gert Faber, Stefan van Oss for giving their feedback, answering me with any questions I had and providing me with relevant materials whenever I needed. Moreover, I would also like to thank TCS employees – Kailas Parande and Megha Jindal who also participated in the study and shared with me their valuable time.

I would like to thank my wife and my parents for their continuous support. Their presence in my life was always an energy source for making ambitious steps in my life.

Contents

Chapter 1. Introduction....7
1.1 Thesis scope....7
1.2 Problem statement....7
1.3 Research question....9
1.4 Document structure....9
Chapter 2. Literature overview....10
2.1 What is technical debt....10
2.1.1 History....10
2.1.2 Definitions....10
2.1.3 Technical debt in practice....14
2.1.4 Layers of technical debt occurrence in systems....17
2.1.5 Literature review findings....17
2.2 Software quality....18
2.2.1 Overview....18
2.2.2 Standards....18
2.3 Technical debt measurement....20
2.3.1 Introduction....20
2.3.2 Reported methods....22
2.3.3 Tools used to estimate Technical debt....25
2.4 Architectural technical debt....29
2.5 Approaches to management technical debt....30
2.5.1 Introduction....30
2.5.2 TD process management....31
2.5.3 TD portfolio management....32
2.5.4 TD ownership....33
2.5.5 Approach by JL. Letouzey....33
2.5.7 Other techniques....34
Chapter 3.Research design....35
3.1 Introduction....35
3.2 Defining the techniques....35
3.2.1 GQM....35
3.2.2 Technical debt template....36
3.2.3 Technical debt taxonomy....37
3.2.3 Sonar metrics....37
3.2.4 Paretto approach....37
3.3 Model application rules....37

Chapter 4. Model description 40
4.1 Introduction 40
4.2 Model parts 41
4.2.1 TD taxonomy 41
4.2.2 Item description 43
4.2.3 Estimation guidelines 44
4.2.4 Calculating and aggregating tool 44
4.3 Model discussion 45
Chapter 5. Application of the model 46
5.1 Introduction 46
5.2 GQM results 46
5.3 Technical implementation 47
5.4 Data collected on technical debt 48
5.5 Evaluation of proposed method 52
5.6 Threats to validity 55
Chapter 6. Conclusions 57
6.1 Summary 57
6.2 Discussion 57
6.3 Contributions 58
6.4 Limitations 59
6.5 Future work 60
Literature list 61
Appendixes 64

Chapter 1. Introduction

Technical debt is a term that was introduced in the developers community over 20 years ago. Growing from an easy to understand and use metaphor that connects product developer and product owner to a mature metric, technical debt (TD) has gained considerable theoretical background for the last decade. One can find an extensive list of comprehensive scientific papers describing various approaches to TD. Among them: best practices (for project managers, developers etc.), measurement techniques, success stories in major brands, tools and techniques, new and adopted by industry and so on.

Most of the literature is targeted at implementing code metrics to produce numerical values representing various aspects of TD; that is natural by following reasons: developers feel code level debt most strongly; code metrics are comparably easier to implement. It can be seen now there is a lack of information how Architecture-level TD (ATD) can be measured, captured and communicated.

1.1 Thesis scope

From a scientific point of view this thesis project will be investigating the theoretical basis to develop a model of architectural technical debt for further practical implementation. Practical studies will be bounded to Internet Banking (IB) systems of ABN AMRO in The Netherlands. Most attention will be paid to architectural technical debt existence, classification and estimation. The whole project can be clearly divided onto four consecutive phases:

1. Theoretical studies, getting familiar with state of the art approaches;
2. Measurement model development
3. Measurement model application and refinement;
4. Evaluation of the approach and conclusions.

1.2 Problem statement

As any mature enterprise organization successfully operating worldwide for decades already ABN AMRO heavily relies on information technologies. The organizational domain of IB and related departments use a vast number of systems to operate. The IT landscape includes systems written in different languages, which have various times of creation (aged from 12 to 2 years old), developed in-house and by sub-contractors. Large amounts of code as well as numerous interconnections and inevitable duplication creates a retention impulse for the whole system. One of the aspects that can be distinguished can be clearly identified as technical or IT debt.

There is a clear understanding by higher management that a strategic approach to restructure the IT landscape is needed – this request was formulated in program called TOPS2020. From a practical perspective this means that currently there is a request in a company to:

1) investigate domains and systems that contain TD ;
2) calculate relative amounts of debt and define most relevant points to rework;
3) provide a strategy to control TD and eliminate it in most valuable points.

Formal application of existing debt approaches cannot provide a complete picture of the TD landscape with enough precision because currently presented TD metrics and techniques are mostly concentrated on code-

level debt (in more details those approaches are described in Chapter 2). While for the company now according to stakeholders the most amount of TD is captured on the architectural level. Considering modern challenges like implementation of cloud infrastructure it becomes extremely valuable to define a detailed TD vision on systems level not on level of code blocks or modules.

Why ABN AMRO needs to measure TD

The aforementioned TOPS2020 program is targeted to describe both high level principles and detailed actions that are needed to take the evolution of the technological landscape of the bank to 2020. Among other strategic ideas the point about technical (IT) debt was introduced.

The current state of the solution regarding IT debt within the bank is that there is a requirement to propose approaches to measure and manage TD. But as it was mentioned before there is no clear method how this can be done on a companywide level. Also because of the variety of interconnected systems in the infrastructure there are no tools that can be set up out of the box.

Development of this method was started as a project by the Multi-channel Services (MCS) department of the IT division and Martin Paris as a project manager. Part of this ambitious and valuable project has become the topic for this thesis research.

Why the Architectural TD domain was chosen

As will be presented in Chapter 2 scientific approaches for ATD are much less developed today. Available case studies usually describe investigations of relatively big but separated systems. Starting from investigating a group of systems in the IB domain the method must be later transformed to a metric that can be propagated on the whole company's IT. Significant part of the systems landscape is planned to be investigated later.

Current state of the ABN AMRO IT systems landscape

Today the company's IB architecture contains dozens of applications providing all kinds of services. More than a decade of developing new modules, integrating 3rd party solutions, adjusting processes to changing laws, regulations and opening business opportunities has led to enormously complex components.

Several years ago ABN AMRO adopted SOA (service oriented architecture) as a main guiding principle for building new applications. Service orientation is an architectural concept that refers to the loose coupling of a service (an abstract resource with a defined job) and its provider (the physical asset(s) that perform the job tasks). A requestor only knows what the service's job is and how to request it. The service itself is the only one aware of its implementation.

1.3 Research question

The study was conducted in a way to answer the following research question:

What techniques for ATD measurement can be applied and how successful are they?

The following list of research sub-questions presents the whole research as a set of logical steps. Each sub question will be answered by specific parts of the thesis:

1. What is architectural technical debt, how can it be measured, and how does it related to other TD measurement techniques?
2. How can ATD best be measured in enterprise IT systems such as those of ABN AMRO?
3. How feasible, useful and reliable are the proposed measurements of ATD in practice?

1.4 Document structure

This thesis is organized as follows: Chapter 2 presents a literature overview of the topic. Chapter 3 contains a description of the research design and its theoretical basis. Chapter 4 is devoted to a description of the developed measurement model. In Chapter 5 the results of applying the model are presented and also know issues and model feedback are discussed. Results evaluation and future planning are in Chapter 6. The appendixes contain various details of the research – definitions, taxonomy tree, collected data tables etc.

Chapter 2. Literature overview

This chapter describes the foundations of technical debt, its connections with software quality, its layers of occurrence and approaches to manage it.

2.1 What is technical debt

2.1.1 History

In the literature on technical debt (sometimes referred as “software debt” or “IT debt”) the most cited source is Ward Cunningham’s report “The WyCash portfolio management system” released in 1992. In this paper the word “debt” was used for the first time to describe the results of violations of good code and architecture practices. It also described the dangerous consequence of a team spending more and more time on new feature implementation (paying debt interest) if earlier violations are not fixed – debt is not repaid.

The problem of software that keeps capturing more and more complexity in itself, captured by Cunningham in the TD definition was mentioned earlier by Meir Lehman[1] in the ‘80s. He posited in one of his laws of software evolution that as a “system evolves its complexity increases unless work is done to maintain or reduce it.”[2]

Several acknowledged software engineers took part in developing scientific approaches to technical debt definition and measurements.

- Ward Cunningham –the creator of wiki, and aforementioned person who first coined the TD metaphor [15] in 1992.
- Martin Fowler – famous practitioner and speaker on software development and team productivity. He described the Technical Debt Quadrant[2] in 2009.
- Israel Gat – head of a consultancy company on software quality, has used the term implementation[3] in his work a lot and wrote a book [20] on technical debt.
- Philippe Kruchten – proposed the layers of TD aggregation in IT systems and participated in formulating other viable concepts of the modern TD ecosystem.
- Steve McConnell - CEO at Construx Software, and famous author of many software development books. His post in 2007 [16] on categorizing and managing technical debt.

2.1.2 Definitions

1. Definition by Ward Cunningham [15]
In the report he says that neglecting the design is like borrowing money.

“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. Every minute spent on not-quite-right code counts as *interest* on that debt. Entire engineering

[1] Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. Proceedings of the IEEE, 68 (9), 1060–1076.
[2] Web link: http://martinfowler.com/bliki/TechnicalDebtQuadrant.html
[3] Gat, I. 2010. Revolution in Software: Using Technical Debt Techniques to Govern the Software Executive Report. Cutter Consortium

organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise."

Considering this both parts of money debt idea can be derived like:

1. Refactoring, it's like paying off the *principal* debt;
2. Developing slower because of this debt is like paying *interest* on the loan.

Later McConnell and Fowler described approaches for TD categorization into distinct types, separating issues depending on whether they were introduced specifically or unintentionally.

2. Definition created by Fowler and McConnel[4]

Martin Fowler's famous post in the blog about the TD quadrant starts with discussing the question whether messy code or bad system design is TD or not. Further on, 4 types of approaches to implementing code are described.

- The *prudent debt* to reach a release may not be worth paying down if the interest payments are sufficiently small - such as if it were in a rarely touched part of the code-base.
- A sloppy and low quality code is a *reckless debt*, which results in crippling interest payments or a long period of paying down the principal.

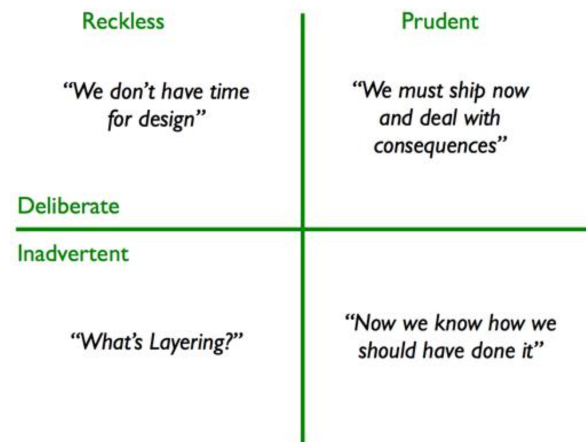


Figure 1. Technical Debt Quadrant by Martin Fowler

This reasoning introduces one of the easiest yet powerful solutions for initial categorizing existing TD – the Technical Debt Quadrant (Figure 1).

3. Definition by Bill Curtis et all.[2]

Curtis provides the following definitions:

- *Technical Debt*—the future costs attributable to known structural flaws in production code that need to be fixed, a cost that includes both principle and interest. A structural flaw in production code is only included in Technical Debt calculations if those responsible for the application believe it is a 'must-fix' problem. Technical Debt is a primary component of the cost of application ownership.
- *Principal*—the cost of remediating must-fix problems in production code. At a minimum the principal is calculated from the number of hours required to remediate must-fix problems in production code, multiplied by the fully burdened hourly cost of those involved in designing, implementing, and testing these fixes.
- *Interest*—the continuing costs primarily in IT attributable to must-fix problems in production code. These continuing costs can result from the excessive effort to modify unnecessarily complex code, greater resource usage by inefficient code, and similar costs.

This set of definitions is also used in "Estimating the Principal of an Application's Technical Debt"[22] in a more shortened version however.

4. Definitions from article by group of authors [25]
Following a workshop at the Software Engineering Institute on June 2-3, 2010, a group of authors agreed on the following definitions:
[TD] Principal – given a particular type of technical debt, the estimated cost of eliminating that debt (e.g., testing, refactoring.)
[TD] Interest probability – the probability that a particular type of technical debt will in fact have visible consequences (e.g., how likely it is that a defect exists in the untested part, or how likely the code in need of refactoring will have to be modified.)
[TD] Interest amount – the added cost of performing maintenance on the part of the system that contains technical debt (e.g., the cost of fixing a defect when it is discovered by a customer as compared to earlier when it would have been detected if testing had been completed, or the extra cost of modifying a component in need of refactoring as compared to the cost of modifying it after refactoring.)

5. Additional TD definitions
Even bigger list of 20 definitions is collected by [33] authors in the list "Explicit and implied definitions of technical debt in academic literature". Some of the useful ones, that can give a better picture on a current topic are enlisted below.

Here most of those descriptions are categorized into two categories considering main groups of stakeholders in software development process: "business guys" as managers and "technical people" as developers. Those two groups usually seen as having different mindset, using different tools for work and even producing different parts of resulting product. But still they have to operate it the same project scope – in time, requirements list and market conditions. That is where the ambiguity and industry value of the term can be seen – it can act as an idea transmitter, common base for building sensible strategy in project development.

Table 1. Additional technical debt definitions

Definition type	Definitions examples
Project management, business side	1. "This pressure [to meet deadlines] can encourage shortcuts concerning code maintenance that lead to the accumulation of technical debt, that is, a backlog of deferred technical problems" (Torkar et al., 2011) 2. "This [simplistically allowing business value to drive development so that architectural soundness is compromised] may lead to increasing maintenance costs and the quality of the end product is undermined" (Heidenberg and Porres, 2010) 3. "The technical debt present is a byproduct of the previous private loan project, as most business and technical decisions were prioritized by the business team" (Davis and Andersen, 2009) 4. "If teams are making decisions to sacrifice quality or maintainability in order to meet those demands [pressures to use fewer resources, hit timelines and show return on investment], technical debt is incurred" (Smith, 2009) 5. "Changing priority to short-term aspects usually contributes to increased technical debt and decreased project quality" (Lindgren

	et al., 2008b) 6. "Just as new firms borrow capital to get started, new software projects borrow 'design capital' (time) to get a product to market. Maintenance problems that ensue are the interest you pay for design errors introduced by schedule pressure" (Lutz, 1993)
Developer, technical side	1. Almost invariably in software projects, developers can be so focused on accomplishing the needed functionality that the software itself grows less understandable, more complex, and harder to modify" (Shull, 2011) 2. "As defined by David Draper, at its broadest, technical debt is any side of the current system that is considered sub-optimal from a technical perspective" (Ktata and Lévesque, 2010) 3. "As they deliver software, teams accrue what the agile community refers to as 'technical debt' in their code. This includes things like bugs, design issues, and other code-quality problems that are potentially introduced with every addition or change to the code" (Black et al., 2009) 7. "Some industry experts view poorly evolvable code as technical debt that can slow down development" (Mantyla and Lassenius, 2009) 8. "Non-agile infrastructure tends to grow old because changes are hard to execute in these environments. This can be seen as technical debt" (Debois, 2008) 9. "I'm acutely aware that when I let my design slide, I'm creating technical debt" (Wirfs-Brock, 2008b) 10. "Complex software systems erode over time. Software systems must be extended, adapted, and modified accordingly as new requirements, constraints, and environments emerge. Developers, however, seldom give these efforts the rigorous consideration of the original design. Consequently, the system decays, resulting in decreased usefulness and increased errors" (Neill and Laplante, 2006)

Considering that debt idea coming from financial knowledge domain, some of the researches [23, 2, 35] used other concepts and methods, related in financial area, for describing debt and activity around it in software development domain. Example can be utilizing Real Option theory by I.Gat [20] to describe opportunities in that manager and developer have on consecutive stages of the project. And how choosing cheaper or faster implementation today can actually cost much more in a month perspective.

Below is a list of financial by origin terms related to technical debt as they are defined by Bill Curtis et all. In article "Estimating the Size, Cost, and Types of Technical Debt" [2]:

- *Business risk* — the potential costs to the business if "must fix" problems in production code cause damaging operational events or other problems that reduce the value to be derived from the application.
- *Liability* — the costs to the business resulting from operational problems caused by flaws in production code. Such operational problems would include outages, incorrect computations, lost productivity from performance degradation, and security breaches.

From a risk perspective, flaws in the code include both must-fix problems included in the calculation of Technical Debt as well as problems not listed as must-fix because their risk was underestimated.

- *Risk* — the potential liability to the business if a must-fix problem in production code was to cause a liability-inducing event. Risk will be expressed in terms of potential liability to the business rather than the IT costs which are accounted for under 'interest'.
- *Opportunity cost* — benefits that could have been achieved had resources been committed to developing new capability rather than being assigned to retire Technical Debt. Opportunity cost represents the tradeoff that application managers and executives must weigh when deciding

6. Technological gap

TD existence is also a matter of project size. Technical debt can arise due to changes in environmental factors that are out of the development team's control even if good decisions may have been made. If the system does not evolve, then new environmental conditions may start creating high interest payments [25].

This may be referred to as Technical Inflation[9] mentioned by Scott Wood - *the ground lost when the current level of technology surpasses that of the foundation of your product to the extent that it begins losing compatibility with the industry*. Examples of this would be falling behind in versions of a language to the point where your code is no longer compatible with main stream compilers.

Philip Kruchten also refers this phenomena as *Technological gap* (refer to Figure. 2) - "This is tech debt that you got by doing nothing, it is just the passing of time, that made the design choice you made now obsolete in the presence of new technology showing up. To keep the product current you may have to close that gap (i.e. adapt to the new technology). So at the time you made the design choice, it was the right choice. 5 years later it is technical debt."

2.1.3 Technical debt in practice

Besides scientific approaches Popular literature also. TD examples are collected from various technical and development blogs and portals.

There can be named plenty of reasons when typical debt generally taken in software development cycles. By studying Below is a sample list of such cases summed up in the table below.

Table 2. Examples of technical debt

Level	Examples	Comment
Architecture level	1. Bad demarcation and rationalization of the IT landscape 2. Inconsistent design approaches 3. Careless mistakes ('we work agile and our code is	Becomes visible for considerably large or/and aged systems when code modules interrelations can harm project properties (robustness, maintainability, future developments costs) more than

[9] Web link: http://www.slideshare.net/lauraxthomson/rewrite-or-refactor-when-to-declare-technical-bankruptcy

	the message') 4. Poor choices of component decomposition 5. Incoherent designs or more complexity in designs than absolutely needed 6. Design choices that turn out to be wrong in hindsight	low quality of each block's code.
Code level debt	7. Violations of coding standards 8. Code duplication 9. Poor or absent comments 10. General sloppiness 11. Refuse or poor usage of OOP, patterns, MVC or other concepts	The most detailed described domain. Mostly because authors being developers first of all apply metaphor on its initial domain.
Test level debt	12. Poor or absent test scenarios and/or test atomization efforts as the critical solution grows 13. Incomplete test coverage 14. Poor test automation	Testing being a valuable part of development process can be underestimated by manages that leads to loss of time and quality for the project
Social (managerial)	15. Intentional debt taken for strategic reasons 16. Debt taken for personal interest (career or the expectation to increase income or to prevent personal reputation damage)	Eventually mentioned, but is quite vaguely defined.

Rationales for accepting TD

As it was stated before, in financial world – debt is not a specifically bad thing to have. Companies consider credits as a tool that can be handy to overcome current market situation or new development challenges. The danger for the future of the project comes when this tool used inappropriately. The same can be applied to software development – some technical debt can help to leverage some current conditions. Specific project implementations are resulting many factors involved in project. Regarding this point of view some of reasons for taking decisions that make software project to implicitly incur technical debt are presented in following table.

Table 3. Rationales for taking on technical debt

Reason	Explanation	Examples [16]
Time to Market	Shortening time to market	When time to market is critical,

	though assuring the debt taken is mitigated in a short time	incurring an extra $1 in development might equate to a loss of $10 in revenue. Even if the development cost for the same work rises to $5 later, incurring the $1 debt now is a good business decision.
Preservation of Startup Capital	Preserving startup capital though assuring the debt gains priority in the requirements backlog	In a startup environment you have a fixed amount of seed money, and every dollar counts. If you can delay an expense for a year or two you can pay for that expense out of a greater amount of money later rather than out of precious startup funds now.
Systems retirement	Delaying development expenses assuring capital is preserved to invest in future technology replacement	When a system is retired, all of the system's technical debt is retired with it. Once a system has been taken out of production, there's no difference between a "clean and correct" solution and a "quick and dirty" solution. Unlike financial debt, when a system is retired all its technical debt is retired with it. Consequently near the end of a system's service life it becomes increasingly difficult to cost-justify investing in anything other than what's most expedient.

Distinguishing technical debt from other issues

It was mentioned by Robert Martin[10] - *a [code]mess is not a technical debt*. Which in other words means that you should not refer bad code practices to technical debt – sometimes sloppy code is just sloppy code that needs to be fixed. Several authors refer to a similar point, especially after talking to practitioners in specific domains one can see that after they get the essence of the metaphor and its flexibility and power. Then they easily fall into the stage when every bad implementation or managerial structure or decision is going to be qualified as technical (business, social, managerial,) debt. Definitely debt is an interesting construct and it has a lot of useful applications but those domains will stay out of the scope of this paper.

[10] Blog post, 09/22/2009, Web link: https://sites.google.com/site/unclebobconsultingllc/a-mess-is-not-a-technical-debt

Technical Debt must be distinguished from defects or failures. Failures during test or operation of the system may be symptoms of IT debt, but most of the structural flaws creating Technical Debt have not caused test or operational failures [2].

Summing up we must state that again, not all incomplete work is debt. It's not debt because there is no need in debt interest payments.

2.1.4 Layers of technical debt occurrence in systems

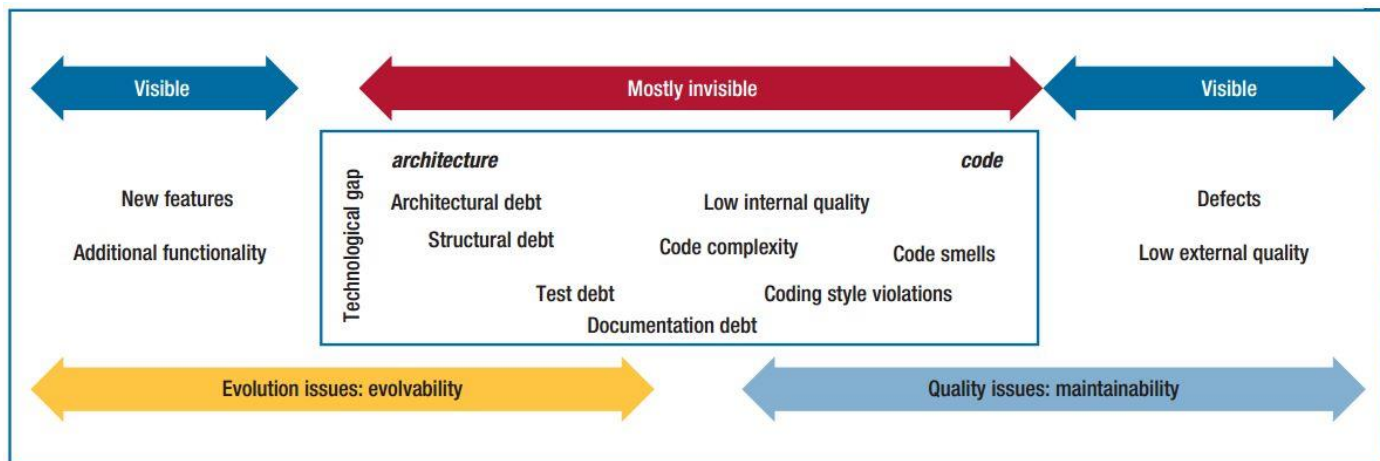


Figure 3. Technical debt domains according to Philippe Kruchten

Philippe Kruchten et all, in [26, 4] introduced the diagram for showing presence of various debt domains across the system. On this picture (Figure 2) one can see the layers that researcher defines from : architectural debt (or structural debt), documentation debt, test debt to code-level debt. The last one is presented by a set of characteristics - code complexity, code smells, coding style violations.

It also can be seen that all the debt domain is behind the visible part of the software properties. It's a very valuable factor especially when it can be seen that IT debt presence directly influences the other parts of the software development process.

2.1.5 Literature review findings

Having investigated quite a wide volume of literature on technical debt and related software quality measurements and approaches the following stats was formed. This also influenced approach used to build our own model for this research.

Table 4. Literature review classification

Type	Item	Articles
Approach/metric	SQALE	[10], [19]
	SIG	[32], [14]
	Matrix	[35]
	Documents analysis	[23], [5], [39], [38]

	CAST	[22], [2]
	Portfolio	[29]
	ISO\IEC 9126	[8], [37], [2]
	Custom	[17], [35], [34], [4], [36], [27]
Article type	Case study	[5], [23], [8], [39], [32], [34], [4], [38], [36], [27], [14], [30], [28]
	Theoretical	[24], [8], [6], [25], [41], [37], [3], [9], [26], [11], [40], [10, [19, [31], [29], [21]
	Interviews	[1], [18]
	Systems research	[22], [2]
TD type	Architectural TD	[4], [36], [27]
	Code level TD	[35]

Below several findings of our review are discussed:

1. TD studies are still having more theoretical discussions, then practical reports with detailed values of debt captured;
2. Practitioners tend to combine existing metrics and propose new calculation approaches based on addressed systems conditions;
3. Documents analysis and questionnaires can serve as powerful method for obtaining quantitative results on TD
4. Sonar tool is widely used, but not that widely discussed in scientific TD literature
5. ATD studies are usually separated from code level TD

2.2 Software quality

2.2.1 Overview

The area of software development has been constantly growing in complexity and impact on economics and society for more than 50 years till now. Being a very practical and also quantitative area of human activity it also developed the approaches to maintain quality of the products delivered and how effective the processes (development) are organized. Considering the context of software engineering, software quality is defined in two aspects[11]:

1) Software functional quality reflects how well it is aligned with a given design (functional requirements) or specifications.
2) Structural quality of the software describes to what extent it meets non-functional requirements (e.g. maintainability). Because they support the delivery of the aforementioned functional requirements.

2.2.2 Standards

ISO 9126 standard

A first edition of international standard for the evaluation of software quality was issued in 1991. It presented[12] six general characteristics that were aimed to give an overview of software quality: functionality, reliability, usability, efficiency, maintainability, portability. Each characteristic is divided in sub characteristics to review.

[11] Pressman, Scott (2005), Software Engineering: A Practitioner's Approach (Sixth, International ed.), McGraw-Hill Education Pressman

[12] Web link: http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749

Table 5. Characteristics of ISO 9126 standard

Metric name	Metric description
Functionality	Up to what extent the software performs as per the requirements and specifications. Testing is used to verify that the requirements are met. This basic of quality factors but can be problematic for large, complex software systems.
Reliability	Reliability is the capability of software to maintain its level of performance under stated conditions for a stated period of time. It is also defined as the probability of failure-free operation.
Efficiency	Indirectly efficiency can be measured by measuring the amount of time (execution efficiency) or storage (storage efficiency) needed when running the software through a particular compiler, under a specific OS, on a designated hardware architecture.
Usability	Usability characteristic is an attempt to define user friendliness. It can be measured in terms of for example physical and intellectual skill required to learn the system or the net increase in productivity over the system it replaces.
Maintainability	Maintainability aimed to define how is easy is software object the to understand, enhance, and correct in future. Sub criteria of maintainability include consistency, simplicity, conciseness, self-descriptiveness, and modularity
Portability	Portability is a set of attributes that bear on the capability of software to be transferred from one environment to another.

Generally this standard introduced a top-down look at software quality and targeted both developers as well as project managers. This also lead to the fact that not all characteristics could be reviewed automatically, for example conformance and compliance relayed on laws and external standard. It has been replaced by ISO/IEC 25010:2011 in 2011.

ISO/IEC 25010

ISO 25010[13] is a product quality model composed of eight characteristics (which are further subdivided into sub characteristics) that relate to static properties of software and dynamic properties of the computer system. Some of the selected characteristics derive from those in ISO 9126. The model is applicable to both computer systems and software products. Those metrics according to authors provide consistent terminology for specifying, measuring and evaluating system and software product quality. They also provide a set of quality characteristics against which stated quality requirements can be compared for completeness. Those metrics are explained on Figure 3.

[13] ISO/IEC25010: Software engineering-System and software Quality Requirements and Evaluation (SQuaRE) - System and software
Quality Model, 2011.

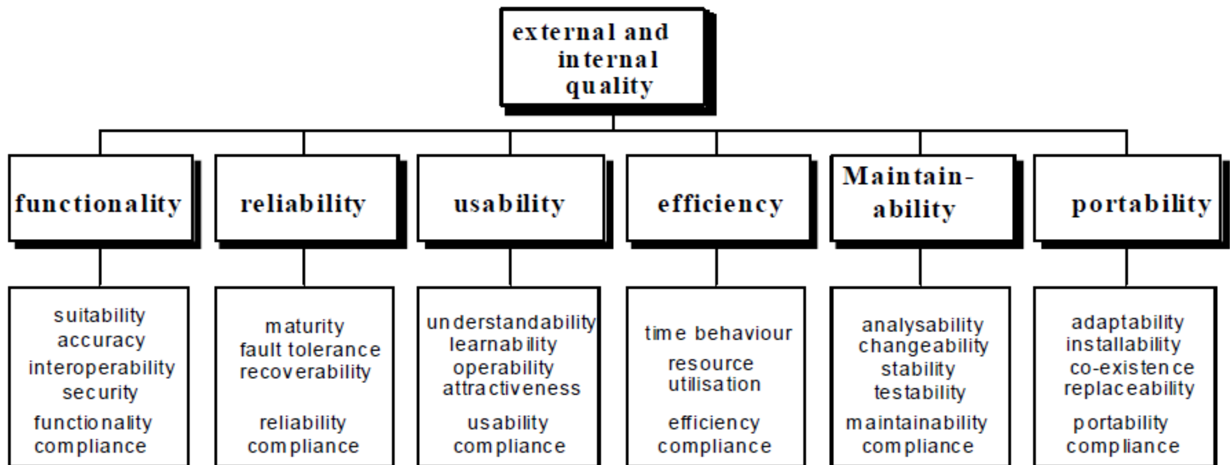


Figure 3. ISO 25010 quality model

Applying of methodology described in the standard can also provide guidance in identifying software and system requirements, design and testing objectives, identifying acceptance criteria and establishing combined measures of quality characteristics. Behind this model, there is an approach called Factor-Criteria-Metric Model[14] which is commonly used in the field of software measurements.

2.3 Technical debt measurement

2.3.1 Introduction

Every day employees involved in the software project have to make decisions. The decisions are related to different levels: developer chooses the most applicable implementation technique, architect is planning what libraries or patterns should be used for next development stages. Also a project manager has to allocate time resources to continue feature implementation according to the schedule while there is also need to decrease amount of shortcuts in the code and temporary architectural decisions.

Software development practitioners have made several attempts to define a quantitative rules and metrics that project stakeholders could utilize to balance speed and productivity versus

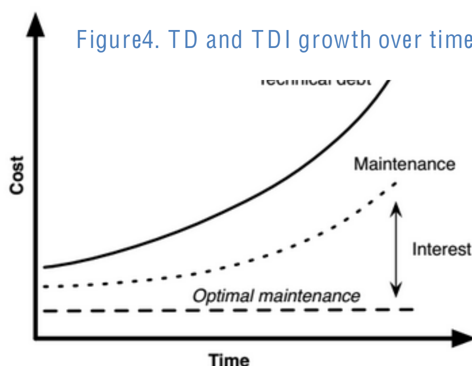


Figure4. TD and TDI growth over time

aches to quantify the amount of debt have led to quite diverse. This situation corresponds to words of al debt – "The tricky thing about technical debt, of le to measure effectively".

[14] McCall J.A., Richards P.K., Walters G.F., Factors in software quality, Vols. I-III, Rome Air Development Centre, Italy, 1977

General representation[32] of the TD growing in the system with the time was presented by Jim Highsmith[15] as on Figure 3.
Introducing a virtual "optimal maintenance" – straight horizontal line for the case when system doesn't change costs across the years one can compare it with real life maintenance presented by dashed graph. Maintenance starts growing even faster if technical debt is incurred by the system.
So in case if development team created big amount of TD in initial stage that makes further project development more and more complicated. The more we're moving forward in the time line the harder the design choices are.
Another outcome for the graph is that technical debt somewhat similar to entropy always grows for the addressed software with time - if nothing is done to handle it, then the situation always gets worse. It also shows that exact TD estimation in applications with high incurred technical debt becomes nearly impossible.

As it was introduced earlier in the definitions part debt incurred by IT systems has two compounds: main debt body (principal) and debt interest (penalty, regular fee). General formula that can be presented as an array.

TD : { **TD***principal* , **TD***interest* }

In hours	Work to be done, to remove the debt	Loss of productivity, Is "paid" every time part of the system is changed

Those two parts values are independent. For example some inefficient source code problems are not likely to cause future maintenance problems or affect the overall quality of the system. In terms of the TD metaphor, the TD principal may be higher than the TD interest being paid on the debt [28]. Or it can be other way around – when code part is changed often, then shortcuts and general poor quality existing in this part will take a lot of additional effort every time. While fixing this exact part (TD principal) can be relatively fast.
There also can be the situation when different parts of TD are aligned with conflicting goals of different stakeholders. For example - the Department1 would like to have well commented code; whereas the development department (Department2) is focused on producing running code, so the Department1 department estimates the interest and the development department estimates the principal [25].
Within one system total debt does not necessarily combine additively, but this can be called super-additively in the sense that taking on too much debt leads a system into a bad, perhaps irreparable state (e.g., of code complexity) [24].
A valuable factor for estimating TD interest for a specific organization is availability of historical data. For instance, by adopting a configuration management system and analyzing source code repositories data we can see the extent to which a component with high coupling and cohesion is less maintainable than other components. Historical data can be useful but might not be available for all the TD [25].

Below is the part describing different techniques to quantify TD principal or TD interest or both.

2.3.1.1 Estimating TD Principal

[15] Highsmith, J. 2009. Agile Project Management: Creating Innovative Products , Addison-Wesley.

Measuring only principal for existing debt in the system differs by approach from estimating both TD parameters. Static software source code at least allows us to estimate the amount of principal based on actual counts of detectable structural problems. Initially it's stated in [2] that principal amount of debt can be calculated using formula:

$$TD_{principal} = N_{\text{must-fix problems}} \times t_{\text{time required to fix}} \times C_{\text{cost for fixing a problem}}$$

But considering that usually in big software systems it never happens that all defects and flaws are removed completely. First, because it's always not enough budgeting. Second constant evolving of codebase makes precise counting all the points that need to be done at least debatable.

In the research [41] authors combined two diverse techniques for identifying debt in the system. They asked different team members – developers, tester, manager about parts of the system that contain most debt and also applied code analysis tools to the codebase of their product. Debt in the system was investigated on several layers, mostly corresponding to domains introduced by Philippe Kruchten [4]: design, code ("defect" term was used), documentation, testing and one different - usability.

Results show that TD knowledge is dispersed and perceived differently by different stakeholders – each participant named different modules where he expected to have most debt. It was also found that code analysis tools show good correlation with spots identified by people. But as code analysis tools mostly can identify debt in code domain, that's why tools can only support the identification of defect and design debt in the project, but not other types of debt that were found by developers. Unfortunately question of comparing several code analysis tools on one codebase was not studied in this research.
This paper also contains estimation about how much time and effort it takes employees to identify the debt occurrences. It's reported that it took participants between 50 minutes and 2 hours to identify and document the TD items for the given system. While answers about difficulty of the task ranged from "easy" to "difficult/high".

2.3.2 Reported methods

2.3.2.1 SQALE method

Developed in France by Inspearit (formerly called DNV ITGS) SQALE method was intended to measure and manage as objectively as possible the quality of source code that projects deliver. The method was designed to be as generic as possible and is applicable to any kind of language and any development methodology. SQALE method is open source and royalty free. However deploying the method for large IT landscapes is a subject for commercial expertise and also there are set of commercial tools using this method for calculations (SonarQube, SQuORE, CodeQ).

Initially defined in[16] SQALE method uses quality models which is based on the ISO 9126 standard. Among include characteristics it has: testability, changeability, and reliability. Specific metric of SQALE method is the so called *remediation indices*. By means of those indices the amount of effort is counted that is required to resolve non-conformities from the

[16] J. Letouzey and T. Coq. The SQALE Analysis Model An analysis model compliant with the representation condition for assessing the Quality of Software Source Code. VALID, 2010.

generic requirements of the quality sub-characteristics. Summing up lower level indices one can get to a general amount of remediation effort (per system or component). Remediation effort can also be translated into financial value, which is an amount of TD in a system.
Among method limitations - there is no clear justification for the remediation indices present. Ideally, the remediation indices should be based on empirical data. Also a SQALE method can only provide estimation of TD amount, but not TD interest parameter [42].

When implementing SQALE method it also requires association of quality parameters with [10] a *nonremediation function*. It is used to quantify all resulting costs of the delivery of one or more nonconformities, such as for example: costs of additional maintenance resources or costs of additional noncompliance related resources (CPU or memory). In other words, the nonremediation function *estimates the penalty* that the product owner might claim as compensation for accepting violations.

2.3.2.2 SIG/TUViT method

Developed by software consultancy company SIG based in Amsterdam, this method is main part of subscription-based service for enterprise customers that provides regular automated release code analysis.

Maintainability sub-characteristic / Product property	Volume	Duplication	Unit size	Unit complexity	Unit interfacing	Module coupling	Component balance	Component independence
Analyzability	×	×	×				×	
Modifiability		×		×		×		
Testability	×			×				×
Modularity						×	×	×
Reusability			×		×			

Figure 4. Mapping SIG/TUViT method charachteristics

This method is intended for the standardized evaluation and certification of the technical quality of the source code of software products. The scope of its main metric [12] - *evaluation criteria* is limited to the internal quality characteristic of *maintainability* and its sub-characteristics including: analyzability, modifiability, testability, modularity and reusability. Evaluation criteria defines 5 quality levels for maintainability represented by a rating from one to five stars. The quality characteristics are determined by measuring a set of software product properties. These properties include following: volume, duplication, unit complexity, unit size, unit interfacing, module coupling, component balance and component independence.

2.3.2.3 CAST report

Researches in this report have studied the density of coding violations with classifying them to several groups of issues like: security, performance, robustness, and changeability of the code. They also introduced several levels of coding violations: high, medium, and low violations. Furthermore, the

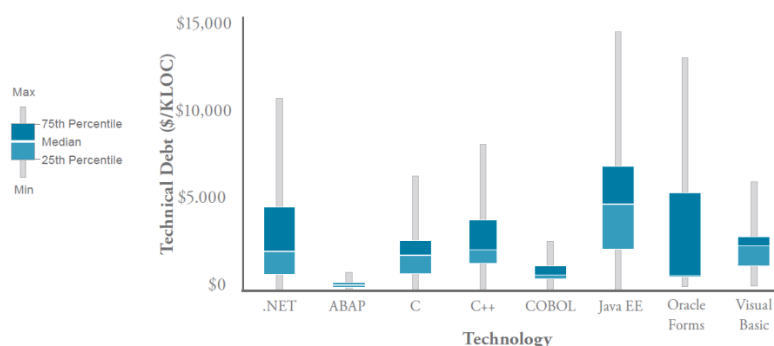


Figure 5. CAST report resulting plot

assumption was made that only 50%, 25%, and 10% of the high, medium, and low violations respectively are actually being fixed [32].
The data in the report[17] was extracted from the Appmarq benchmarking repository (source codes of companies – clients maintained by CAST), which contained 745 applications from 160 companies in 14 countries, comprising 365 million lines of code at the time of the analysis.

In [22] researchers have conducted an investigation of TD-related parameters for almost 700 applications with total 357MLOC. The analysis was performed using CAST's AIP[18] which analyzes an entire application using more than 1,200 rules to detect violations of good architectural and coding practice. The technique include following steps:

- parsing an application's entire source code at build time to produce metadata metrics;
- those metrics then are sent to evaluation module, which applies set of 1200 quality rules that can capture both bad coding practice and architectural miscounts;
- detected violations are grouped in several types and they are processed to form the output metrics;
- metrics area grouped in following categories: *robustness, performance efficiency, security, transferability and changeability*. They are based on ISO/IEC 9126, however changed due to several reasons.

Each of the violations is weighted according to its severity level – from low to high. In real life projects it's never possible to fix all point that are needed to be fixed. That's why additional criteria - (% to be fixed) is introduced. It means that desired level of systems quality change can be set up – for example, 100% of high-severity violations and only 10% of low-severity violations must be fixed to achieve significant results. By using this each product owners and managers can set specific reduction targets based on strategic quality priorities. Those measurements are references in the paper as *estimations* (estimation 1, estimation 2, estimation 3), ranging from more conservative to best possible. Including all of the above, the following formula for estimating TD principal is used:

$$TPD_{in\ hours} = (\Sigma_{high\text{-}severity\ violations} \times (\%\ to\ be\ fixed) \times (average\ hours\ needed\ to\ fix)) + (\Sigma_{medium\text{-}severity\ violations} \times (\%\ to\ be\ fixed) \times (average\ hours\ needed\ to\ fix)) + (\Sigma_{low\text{-}severity\ violations} \times (\%to\ be\ fixed) \times (average\ hours\ needed\ to\ fix))$$

As a result it was found that for JavaEE applications (which present up to 60% of analyzed software modules) that :

- most TD is incurred in following categories: transferability, changeability and robustness
- wide range of costs is presented: from $0.23 per LOC to $253.03 per LOC

This research gives a good insight not only for team willing to estimate the TD incurred in existing codebase. It also provides good approach for project managers trying to set up a communication with business using understandable criteria list and set of progress goals – estimates.

2.3.2.4 Technical debt template

---

[17] CAST worldwide application software quality study: Summary of key findings, 2010.
[18] Application Intelligence Platform, web link : http://www.castsoftware.com/products/application-intelligence-platform

An example of formalized questionnaire form was proposed in [7] for defining the properties of each TD issue in the system. Each occurrence id described by set of meta-data parameters, such as location of a shortcut in system module and/or file, release index, date added and employee details and exact description of an issue. Debt amount is captured by estimating probability, interest and principal of each issue. Measurement of amount can be done in hours or using more indirect scale: from “high” and “medium” to “low”.

ID	37
Date	3/31/2008 (Release 3.2)
Responsible	Joe Developer
Type	Design
Location	Method ***calculateStateTax*** in Module ***TaxCalc***
Description	In the last release, Joe added method ***calculateStateTax*** quickly and method is overly complex and not documented.
Estimated principal	Medium (medium level of effort to refactor and clean code)
Estimated interest amount:	High (it will be costly to make changes to the method in future, especially by other developers)
Estimated interest probability	High (it is very likely that this methods needs to be changed with each future release)

Figure 6. Technical debt template example

Another option is to estimate debt value in hours, days, weeks or months for each issue.
This template was also used in [39] to collect feedback from product development team. According to this paper it takes employees around 15 minutes to fill in each item. So measuring debt spots across all the system can be quite time consuming in case of large systems.

2.3.2.5 Open source projects

Techdebt.org

In 2013 open technical debt collaborative and open benchmarking dashboard was launched on techdebt.org domain. The site aimed to provide several metrics regarding the technical debt for a large panel of applications. The idea behind it was to present continuous results on code quality of a wide range of open source projects using open-source plugin for Sonar. So that developers could compare the quality of their code with hundreds of other projects from the open source and software industry. Unfortunately now the website is unavailable.

Drupal CMS

Also an interesting research of technical debt incurred by open source CMS Drupal was published in one of the blogs on this platform[19]. In it author analyses queries results to the main repository, comparing amounts of reported issues, critical bugs and fixes in during the development of expected new 8th version. As a result of fast incurring debt in the code modules, the release team first had to freeze feature and it’s still unclear when new version can be released.

2.3.3 Tools used to estimate Technical debt

There are a lot currently available on the market proprietary and open source tools[20] for static analysis. They can be selected for specifically language or some provide set of language-

[19] Web link : http://xjm.drupalgardens.com/blog/technical-debt-drupal-8-or-when-will-it-be-ready
[20] Web link : http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

specific parsers. Using ASA tools during the development is considered a good practice for long time.

2.3.3.1 Dynamic code analysis

Typical dynamic code analyzers profile your system and monitor its health. Both execution time and memory usage profilers, figuring out number of database transactions per request, the average size of an user session object, etc. require the system to be under a load comparable with the intended in production environment. Dynamic analysis tools often instrument the code to add tracing of method calls, catching and notifying about exceptions, and any other statistics they collect.

2.3.3.2 ASA tools

The basic principle is analyzing code structure without executing it. This approach is generally used to find bugs or ensure conformance to coding guidelines. The classic example is a compiler which finds lexical, syntactic and even some semantic mistakes.
Static code analysis provides following advantages:

- Full code coverage. Static analyzers check even those code fragments that get control even very rarely while parts of code usually cannot be tested through other methods.
- Static analysis doesn't depend on the compiler that is used and the environment where the compiled program will be executed – helping to find hidden errors that can otherwise reveal themselves only a few years later.
- ASA tools also can give recommendations on code formatting, even some static analyzers can check if the code corresponds to the coding standards used across the project
- Variety of metrics computation - a numerical value of some property of software or its specifications. There are lots of various metrics that can be computed with the help of certain tools.

Static analysis usually presents poor results regarding diagnosing memory leaks and concurrency errors. The actual code execution is needed to detect such errors. Dynamic analysis tools are more efficient way to detect those types of errors.
A static analysis often give so called false-positive - it means that the code can actually be quite correct. So it takes a developer to understand if the analyzer points to a real error or it is just a false positive.
While being super-fast analyzing hundreds lines of code per fractions of second automated analysis tools don't find every issue, but rather search for common types of errors and flaws. It’s impossible for an automated tool to check if the code has been implemented since it don't understand how the logic is supposed to work. The source complexity also increases the ASA demand on system resources - as they exponentially increase the potential paths to be checked.

Researchers in [8] conducted several case studies against following groups of software quality characteristics (selected from ISO standard): Functional suitability, Performance efficiency and Maintainability. According to the results in all cases tools used (Resharper and FindBugs) point right places where the refactoring should be done to improve corresponding characteristics. Unfortunately this paper didn’t present any quantitative results. But the statement is made that ASA should be used to target specific sites in the software to decrease TD interest.
Using any solution that provides metrics differentiating logical parts of the project (files, modules or bigger blocks) can give insights to the team

2.3.3.3 Specific solutions

A. DebtFlag

Is a model application developed in [6] which is implemented in a Eclipse IDE plugin accompanied with web-application UI. By getting access to development environment it captures TD in set of recorded related to specific parts of SUD. Each note has a set of attributes such as description, time and date, author etc. A DebtFlag element is a link between a technical debt observation and an implementation part defined by the technique. For example a package, a class or a method in object-oriented technologies. Currently it's targeted to the Java environment. DebtFlag facilitates knowledge about the class and module dependencies to produce an overall map of existing TD issues in the system. Threshold level and a set of other parameters are used to limit the dependency propagation model. By doing so tool implements so called *micromanagement of TD* for the team, on the level of each developer, by maintaining the presentation of existing issues or TID (Technical Debt Items). Each TDI has the following events: create, modify, resolve – handling the lifecycle of records. Among its features authors name:

- Documenting the real project code structure of emerging TD - developers can track changes of TD amount and can make better decisions (ex: not to rely on too underdeveloped parts of code with large TD incurred)
- All debt records are maintained manually – that gives more accurate and targeted detail level. Whether the TD is intentional or inherited from previous stages of project development better reasoning of future steps can be made.

Those features are also the limitations the project has in its current stage – human time resources are consumed on enumeration of issues and no additional derivative logic base on code analysis can be implemented.

B Resharper[21], FindBugs[22] and others

Those tools mentioned in [23] can be used for analyzing software code for producing metrics that can point out TD spots. It's also a code-level TD investigating solution. By applying large sets of code quality rules (code smells) overall indexes on code quality are calculated.

FindBugs is a byte code analyzer only targeted for Java code, it scans source code for possible bugs, applying bug patterns[23]. First software release was in 2006, it's developed by the University of Maryland. is List of found bugs is a ranked list on a 20-point scale. The lower the number, the more impactful could be the bug. Sometimes FindBugs is used in a combination with PMD (also ASA tool) because it's better in cursory check on best practices. It can be also included in a form of ItelijIDEA or NetBeans plugin.

2.3.3.4 SonarQube (formerly Sonar)

This platform is one of the most popular world known solutions for enterprise software quality measurement. It contains parsers for 20+ different languages, and a plugin-based enhancement system.

Many dashboards with key metrics are available out of the box. And also possibilities to extend core functionality by using a plugin systems are available. Sonar applies wide set of

[21] Web link: http://www.jetbrains.com/resharper/features/code_analysis.html

[22] Web link: http://en.wikipedia.org/wiki/FindBugs

[23] Web link: http://findbugs.sourceforge.net/factSheet.html

software quality heuristics like code duplications, coding standards violations, lacks of test coverage, potential bugs spots, module complexity etc. As part of its analyzers, Sonar core uses tools to find coding rules violations (PMD, Checkstyle), detect potential bugs (Findbugs) and measure coverage by unit tests (Cobertura, Clover). But what makes Sonar truly unique is Squid, its own code analyzer that not only parses source code but also byte code and mixes the results. It can be considered as high-level project analyzer. Sonar has a flexible architecture that consists of three main components:

1) A set of source code analyzers that are grouped in a Maven plugin and are triggered on demand. The analyzers use configuration stored in the database. Although Sonar relies on Maven to run analysis, it is capable to analyze Maven and non-Maven projects.

2) A database to not only persist the results of the analysis, the projects and global configuration but also to store historical data for analysis.

3) reporting tool to display code quality dashboards (web interface) on projects, hunt for defects, check history of changes and to configure analysis.

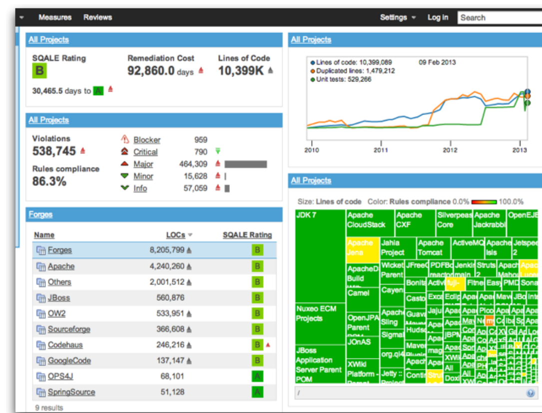


Figure 8. SonarQube dashboards view

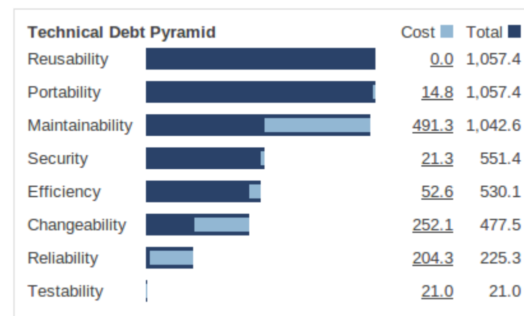


Figure 7. SonarQube metrics presentation

When analysis is run through a Maven plugin, Sonar can also be launched in continuous integration environments. While some researchers report Sonar TD metrics calculator to result unbelievably huge, digits [34] it's still a useful tool for thanks to its core software quality metrics.

In another case study research [5], authors describing implementation of application working with MS Exchange Server in 2006-2008. Developer team has made a decision of implementing WebDAV protocol in 2006 already knowing that newer version MS Exchange 2007 will not be compatible with it. This was done with intent to shorten time to the market. Indeed target was reached - first version was deployed to customers in 2007. Later the same year industry started little by little migrating their systems into newly released MS Exchange 2007. This created a gap in functionality that had to be fixed by implementing new version support, which was done. Sufficient efforts were taken to rewrite some application parts. Authors used unified code count metrics on each release step to estimate the amount of efforts needed form the teams to overcome previously wrong decisions. This paper provided a good modeling approach of how wrongly estimated amount of incurred TD significantly influenced the costs of forgoing development.

2.2.2 CodeQ Quality Investment[24]

[24] Web link: http://codeq-invest.org/

Figure 9. CodeQ quality investment schema

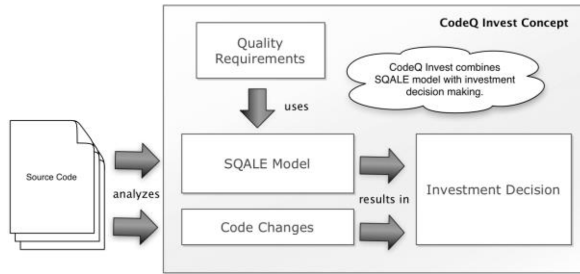


Figure 10. CodeQ schema

This methodology was not widely described and can be classified as experimental or derivative from SQALE. CodeQ tries to connect directly software quality and financial investment approach.
While the SQ part relies on SQALE method, the other parts tries to determine which problems should be fixed to gain an immediate productivity advantage.
First step is to define quality requirements (e.g. code coverage should be greater than 80%) and estimate how long it would take to fix the violation of such a requirement. Next the costs estimation needed - how much is will cost if we to leave the violation untouched. Remediation costs and the latter the non-remediation costs are include. A “profit” is derived from those two measurements - the time/money that can be saved when the violation is fixed.
Such an approach brings reporting IT debts even closer to business language level. Team and product owners can think in terms of ROI and profits when discussing internal software quality.

2.4 Architectural technical debt

What is architecture

According to ISO/IEC 42010 architecture is *fundamental concepts or properties of a system in its environment embodied in its elements, relationships and in the principles of its design and evolution.*
Referring back to scheme by Philippe Kruchten (Figure 2) architecture quality has almost no or little externally visible value “customer value”. While iterations planning during the agile development process is driven by as much “customer value” as possible. That is why also proper architecture planning and architecture rework are often misguided. This can be another point why is that TD at the architectural level it is less researched until now.

When significant architectural change is needed, small, local refactoring efforts cannot compensate for the lack of a coherent system-wide architecture. In the context of large-scale, long-term projects, there is distinction between code-level and architecture-level abstractions, especially when it comes to relating these to a global concept such as debt [25].

In the case study research [4] analysis of development process was performed. Given system - DRNEP[26] was developed primarily at the University of British Columbia, with collaborators in various parts of the world. System architecture consisted of core and distributed simulator modules. Authors considered 2 different architectural approaches and hence 2 various development paths:

1) First called *deliver soon*, this approach assumed incremental addition of new modules, each time adding various ad hoc adaptors and translators for them to fit communicate with core.

[26] Disaster response network-Enabled Platform

2) Second, targeted to *reduce rework and enable compatibility* – was in developing canonical data model, and using an ESB[27]

Development process was analyzed on a period of 4 releases. Initially choosing 1st lane took less time to implement and implementation costs were lower that for the 2nd one. But from one release to another implementation costs for *deliver soon* approach tend to stay the same high or even grow, compared with second approach – where initial cost where relatively high but later were kept on the same low level with zero rework costs.

As a result by 4th release cumulative cost release cost of 1st solution became 55% higher comparing with architecture-wise implementation. Comparing those two paths gives a good insight into the challenge of balancing rapid deployment and long-term value and a value of architecture in dealing with technical debt.

Dependency diagrams for both cases are presented in Fig.9 – it can be seen that 2nd one is also much easier to modify and to understand for people outside the project – managers and new developers joining the team for example. Dependency analysis showed following numbers to compare: 94 vs 116.

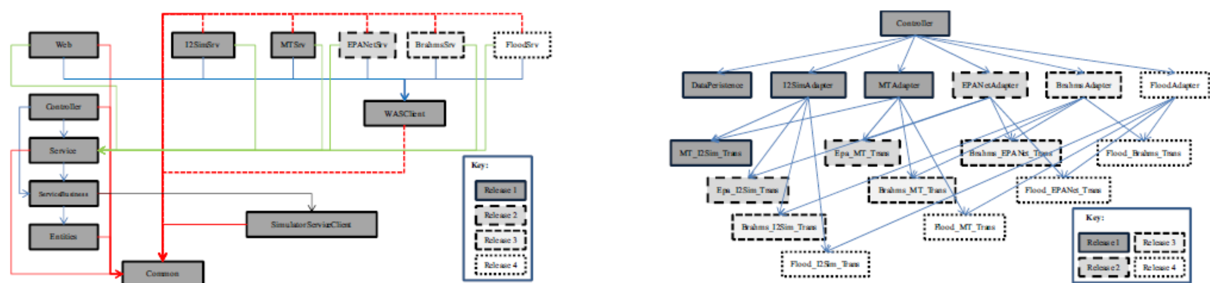


Figure 11. Dependency diagrams for 2 different architecture approaches [4]

This research clearly shows the value of the delivered features compared with the impact of rework costs. Architectural debt can be insensibly incurred by the project later negatively affecting the agility of development process. This should be taken into account by stakeholders in decision-making related to delivering a product.

2.5 Approaches to management technical debt

2.5.1 Introduction

Technical Debt is a metaphor, which is helpful in two cases: while thinking about how to deal with design problems, and also - how to communicate that thinking to the involved stakeholders.

In business dictionary[28] term "management" is defined as follows - *The organization and coordination of the activities of a business in order to achieve defined objectives. Management is often included as a factor of production along with machines, materials, and money...* Applying this to a certain process management can be described as organization and coordination of the activities related to this process to comply with set up objectives.

---

[27] Enterprise service bus

[28] Web link: http://www.businessdictionary.com/definition/management.html

For example in the world of finance and banking debt can only be valid when it is rationalized. The debt investment returns value faster than the negative interest rate taken from the debt over a short time.
An example for rationalized reasons to choose for debt could be to strategically shorten time to market for the proposition expecting to generate revenue (over time) exceeding the debt taken including interest. Not rationalizing the debt choice introduces the risk your debt strategy turns into a vicious circle increasing the debt and can ultimately leads to bankruptcy. Literally any activity taken that does not result in added value now or in the future has nothing to do with debt; this is simply cost and must be seen as waste.

The above example illustrates taking debt can take positive effects as well as negative. It also clarifies it would be wise to manage debt. Technical debt works alike the above example. Technical debt should therefore only be taken if it is proved it pays off in a short time. Therefore technical debt should only be taken for a short time as interest rates grow for the time the debt is taken.

Historically technical debt measurements and reasoning across the industry tended to focus more on programming aspects of software delivery and left out full software development lifecycle.
Each type of TD can be managed and monitored using different tools and approaches.
Focusing on managing each type of software debt simplifies creation of overall strategy that promotes hostile perspective.

To provide effective management one needs to collect reliable information. Hence from the perspective of technical debt management, the goal of identifying and measuring technical debt is to facilitate decision making [31]. From this point one can see two management approaches that will be discussed further:

1) process management;
2) portfolio management.

From time perspective as well there can be short-term and strategic TD-related decisions. The technical debt concept is gaining traction as a way to focus on the long-term management of accidental complexities created by short-term compromises [25]. The same point is supported by Robert L. Nord et all. in [4] stating that technical debt management is about navigating a path that considers both value and cost, to focus on overall return on investment over the lifespan of the product.

One of the questions for long-term and big projects and is how to establish TD awareness and continuous management. In other words - how to TD can be made more visible for all involved parties.

2.5.2 TD process management

Business departments have to govern the processes based on positive outcomes that can obtained. Since the TD metaphor was introduced, there are several key points were described by the IT community to convince management that the changes will help the company:

- Better architecture will allow the team to add new features more quickly;

- Demonstrate that by continuing along the current path (architecture, development) they're putting themselves into a corner and the price to get back on the effective track can be extremely high.
- Providing examples of changes that are very expensive to make in the current system, but which would be simple and cheap with a better design;
- Keeping track of time spent maintaining legacy code vs. adding saleable features. This can also be a Help them explain to existing and future customers that while the existing system was pretty good, the new, modernized architecture will allow many great new improvements, better reliability, and so forth.
- Mitigate the risk associated with the changes you propose. Managers are risk averse, and sweeping changes to an existing system seem inherently risky.
- Prioritize modernization of the various components according to cost and benefit, and make sure that management agrees with your priorities.
- Track the progress of the modernization effort. Show benefits as soon as you reasonably can, but remind all parties that there's more work to do.

Basically rule of thumb for the business concerning technical debt sounds as “Pay now or pay more every day”.

2.5.3 TD portfolio management

The research [23] provides another angle on management issues related to TD. Taking financial way of perceiving TD Ken Power proposes using financial *Real Option*[29]*s* definition for describing teams activities. The same as investor’s money pool is limited, organization (whether business unit or entire firm) has a finite amount of (work hours) capacity to invest in its overall product development efforts. And if the team during initial stages of the project is only targeted on implementing new features this hence gives excellent results in short-term perspective. While on the scale of several releases one can see that TD pay-off activities will become more and more valuable – taking more and more team’s time. Many teams fail to invest adequately in managing and reducing technical debt. Not a lot of details of case study are provided, but it’s stated that teams are using up-to date development approaches based on Scrum, Kanban, XP. For those kinds of teams implementing efficient TD management is especially valuable because the backlog tend to contain only user-stories without cases related TD identification and removal [4].
But the concept should not be taken too far away from the initial definition – not to create a paper monster which is far from real work processes taking place. As Steve McConnell states in his interview[30] - “I have had the experience of software companies taking the metaphor too far, saying for example that they would like to track their technical debt on their balance sheet as a numeric value. The technical debt concept is a house of cards: the numbers we are using to represent technical debt are only estimates on how much path A would take versus path B. Some organizations are good at calculating this estimate, but others are fair, at best. Looking at the foundation of this house of cards and understanding what the technical debt notion means, I think that it is a helpful concept to start a discussion rather than giving out specific numbers and putting them into a spreadsheet”.

---

[29] “An alternative or choice that becomes available with a business investment opportunity. … Taking into account real options can greatly affect the valuation of potential investments” - definition from Investopedia: http://www.investopedia.com/terms/r/realoption.asp
[30] Web link: http://www.ontechnicaldebt.com/blog/steve-mcconnell-on-categorizing-managing-technical-debt/

2.5.4 TD ownership

Calculation of incurred TD across company's IT-assets can be converted to countable amount of money. Since then an important question is who in the business structure of the company taking responsibility for the technical debt taken for this or that solution. This should influence budgeting planning in a way that bad resources could be allocated to eliminate costly TD parts as soon as possible.

Usually it is assumed when the company is not too big and departments structure is not very diverse then the development department takes this responsibility. As they are who have mandate to decide what approach to take. But in case when the company is bigger and products are hence more complicated it can also happened that debt is created and distributed across several departments. Consider the following example:

- Business department – "pushes" the developers to speed up the process according to marketing-related deadlines
- Product development department uses fast-and-dirty solutions, and shortcuts. They probably implement not optimal and costly solutions to deliver another version of product as soon as possible. Afterwards they need to switch to next planned targets – while the debt is taken here.
- Maintenance department – later as a result has to deal with low quality code and architecture for a long time and lacks sufficient resources for that (the debt has to be paid here)

Scattering financial responsibility over several departments where TCO costs for a solution are being managed by different stakeholders (business owner, IT, development department, operational party, management) and incentives tend to defuse insight on TCO. Financial responsibility and accountability of one only party for chosen technical solution details. This party should have the proper mandate to take upon that responsibility so it can be taken accountable. And it's logical to have a business as a responsible point for the decisions made. But then there comes a point that a proper and reliable TD measurement and communication tool must be implemented.

2.5.5 Approach by JL. Letouzey

Jean-Louis Letouzey[31] proposes the following 7-steps approach:

1. Define what creates TD in your systems
2. Define how to calculate TD in your systems
3. Set goals at organization and/or project level
4. Monitor TD against goals
5. Compare TD across applications, versions, projects, 3rd party contractors
6. Analyze your existing TD (age, location, impact)
7. Set pay down goals and prioritize them

It's more a matter of integrating business considerations into technical decision making and vice-versa.

---

[31] Jean-Louis Letouzey, The SQALE method: Meaningful insights into your Technical Debt, Web link http://www.slideshare.net/Letouzey/the-sqale-method-meaningful-insights-into-your-technical-debt . Another even simpler to remember approach was formulated out of word debt: Discover, Estimate, Break Down and Task & Track.

all decisions made in this context are business decisions. However, businesses have been flying blind for a long time when it comes to technical debt. The metaphor therefore helps the business and technical staff have a concrete and open conversation about the technical path to follow that will make the most sense for the business.

2.5.7 Other techniques

One can find other approaches towards facilitating better TD tracking and introducing general awareness about the existing debt.

Utilizing defect tracking system

One of the options described [16] – TD is captured and tracked via defect tracking system. Each time a debt is incurred, the tasks needed to pay off that debt are entered into the system along with an estimated effort and schedule. The debt backlog is then tracked, and any unresolved debt more than 90 days old is treated as critical.
For another company [16] IT debt listing is included as part of its Scrum product backlog, with similar estimates of effort required to pay off each debt. In this approach also a size of defects is regulated by following principle - if the shortcut the developer is considering taking is too minor to add to the debt-service defect list/product backlog, then , it's too minor to make a difference hence it’s better not to take that shortcut.

Chapter 3.Research design

This chapter describes the steps of the research and summarizes the approaches and methods we used to build the model.

3.1 Introduction

After literature analysis the following steps were preformed:

1. Defining the techniques that can be used to build the model (Part 3.2)
2. Describing the rules according to which the model can be adjusted (Part 3.3)
3. Model description (described in Chapter 4)
4. Applying the model – collecting data (described in Chapter 5)

3.2 Defining the techniques

Among several techniques that are used in software quality studies to build reliable metrics adjusted to specific conditions include the goal question metric (GQM) approach [13].

3.2.1 GQM

For defining the model application approach the Goal Question Metric approach was chosen. GQM is a top-down approach when researcher first defines top-level requirements – goals, then follows set of questions and metrics to measure them.

A bottom-up approach will not work in our case because there are many observable characteristics in IT systems (e.g., time, number of defects, complexity, lines of code, severity of failures, effort, productivity, defect density), but which metrics one uses and how one interprets them it is not clear without the appropriate models and goals to define the context.

Due to the way that the project was developing GQM was applied in reversed order – metrics we've used were later retrofitted using this method. Thus, GQM was not so much used for constructing the model but for validating that it was well-constructed.

About GQM

The Goal Question Metric approach is based upon the assumption that for an organization to measure in a purposeful way it must first specify the goals for itself and its projects, then it must trace those goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals. Thus it is important to make clear, at least in general terms, what informational needs the organization has, so that these needs for information can be quantified whenever possible, and the quantified information can be analyzed to whether or not the goals are achieved[13].

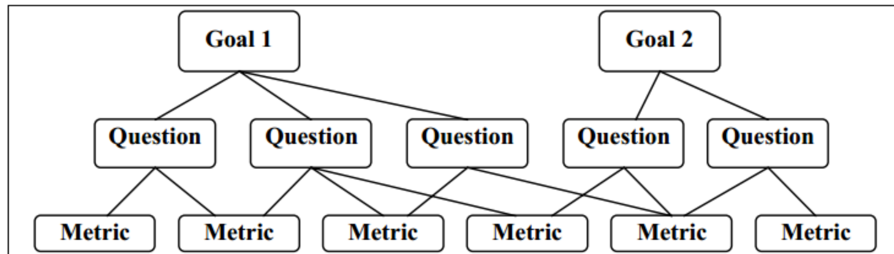


Figure 12. GQM approach schema

For existing 3 layers

- Goal is defined for an object, for a variety of reasons, with respect to various models of quality, from various points of view and relative to a particular environment.
- Set of questions is used to define models of the object of study and then focuses on that object to characterize the assessment or achievement of a specific goal.
- Metrics, based on the models, is associated with every question in order to answer it in a measurable way.

One can use following steps to implement the approach:

1. Develop a set of corporate, division and project business goals and associated measurement goals for productivity and quality
2. Generate questions (based on models) that define those goals as completely as possible in a quantifiable way
3. Specify the measures needed to be collected to answer those questions and track process and product conformance to the goals
4. Develop mechanisms for data collection
5. Collect, validate and analyze the data in real time to provide feedback to projects for corrective action
6. Analyze the data in a post mortem fashion to assess conformance to the goals and to make recommendations for future improvements

3.2.2 Technical debt template

Each issue is described by a set of characteristics derived from TD template [7]. We have additionally modified it to fit our local requirements. And also we added type of debt by intention (see M Fowler quadrant) – as a part of metadata.

3.2.3 Technical debt taxonomy

As there are various approaches to classifying TD (in more details it was described in Chapter 2) issues. For our model we decided to combine several of them.

1. Type of debt by system layer (by P. Kruchten [26])
2. Category within each system layer (TOGAF)

3.2.3 Sonar metrics

As we have code-level TD as one of the domains in our model we decided to collect data no that using Sonar tool. It's planned that results obtained could be useful for reasoning about the quality of the model.

3.2.4 Paretto approach

To make initial data collection faster we've taken a decision to apply 20/80 principle. This was done by using it in both dimensions:

1) Amount of systems for which technical debt that was measured. We have selected 5 systems out of more than 100 applications based on professional opinion of the practitioners who expect those to be most impactful in terms of variety of data collected and checking model feasibility.
2) Decreasing amount of ATD subcategories that were derived from TOGAF. We picked 11 most valuable out of 34 based on professional opinion of the practitioners.

3.3 Model application rules

Model we have defined contains several points that need to be adjusted according to requirements of the research and local conditions. We have developed the following guide for customizing the model before actually applying it for gathering data.

Table 5. Stages of customizing the model before applying

#	Title	Description	Examples and sources
1	Select TD domains	Model in its initial state is targeted to capture all domains of technical debt. But it can be customized to leave only relevant ones – depending on the focus of the project where it will be used.	Available list [26] : 1. Architecture 2. Documentation 3. Testing 4. Code 5. Technological gap
2	Adjust categories	For each of the selected high level domains perform additional categorization providing case-specific taxonomy. This will form the TD landscape that will be investigated. For each group prioritization may be needed in case if the taxonomy tends to be too wide and complicated. First level impact issues can be checked as a priority others will follow.	
2.1	Architectural	Depending on what methodology is used in company categories could be derived	Available list: • TOGAF

		from them	• Zachman EF • FEA • See others[37]
2.2	Documentation	Enlist what types of documentation are used in company. Include also the ones that needed but not in place yet.	Available list – see model taxonomy
2.3	Testing	Enlist what types of testing are used in company. Consult the employees who take care of testing - about specific problems and critical points that are missing.	Available list – see model taxonomy
2.4	Code	Select a tool for detecting code-level debt – structural violations, code smells, complexity etc.	Available list: 1. Sonar 2. Resharper 3. FindBugs 4. Subscription services by SIG, CAST, etc.
3	Adjust TD template	Initial template [60] contained 6 fields. In initial view of our model we have 8 fields. One should think carefully what indicators he/she wants to capture for each TD issue. TD principal and TD interest estimation is expected to be done in hours. Due to the fact that it is hard to estimate it so precisely the following scale of relative times can be used: Hours – Days – Weeks - Months	Obligatory fields: • TD principal estimation • TD interest estimation • Date added • System name • Functional part or module Optional fields: • Employee name/id • Type of debt (from TD quadrant) • System owner • Issue description
4	Tool for collecting data	This can be done by several ways, depending on complexity of taxonomy.	Available options: 1. Using advanced Excel sheet 2. In form of online survey 3. Special TD dashboard
5	How-to manual for data collection	It's targeted to provide guidance for colleagues who will be adding information into this TD database. Depending on the complexity of the taxonomy and fields in TD template it can be quite simple or rather complex. Measure the time on how long it can take to fill in the form for one person. Be sure to allocate sufficient time in employees schedule.	Manual from current paper can be used as a basement [see Appendix]

[37] Web link: http://pubs.opengroup.org/architecture/togaf8-doc/arch/chap37.html

6	Allocate systems list and participants list	If there is sufficient amount of employees who can take part in the survey it's strongly recommended to have overlap. In a way that several participants provide feedback on the same system – this will help to highlight all possible problem spots. Participants can be assigned to specific TD domain according their specialization.	For example – testers should fill in the part about test coverage on selected systems, not answering questions on architecture.

As a result of the aforementioned steps one should have following deliverables:

1. Tailored model taxonomy and issue template
2. Accessible tool for collecting data
3. Research participants (both employees and systems) listing and planning

Chapter 4. Model description

This part of the paper is devoted to model description. It shows the model we have developed, what are the constituent parts and why the decisions were made on each step of the research.

4.1 Introduction

The model itself consists of the following building blocks:

1. A taxonomy of technical debt types and subtypes
2. A template for recording technical debt items that occur in a specific system
3. Guidelines for estimating the size of technical debt items
4. A tool for calculating and aggregating technical debt items

Each of those model elements will be described in this chapter. An overview of the model is presented in Figure 12 below.

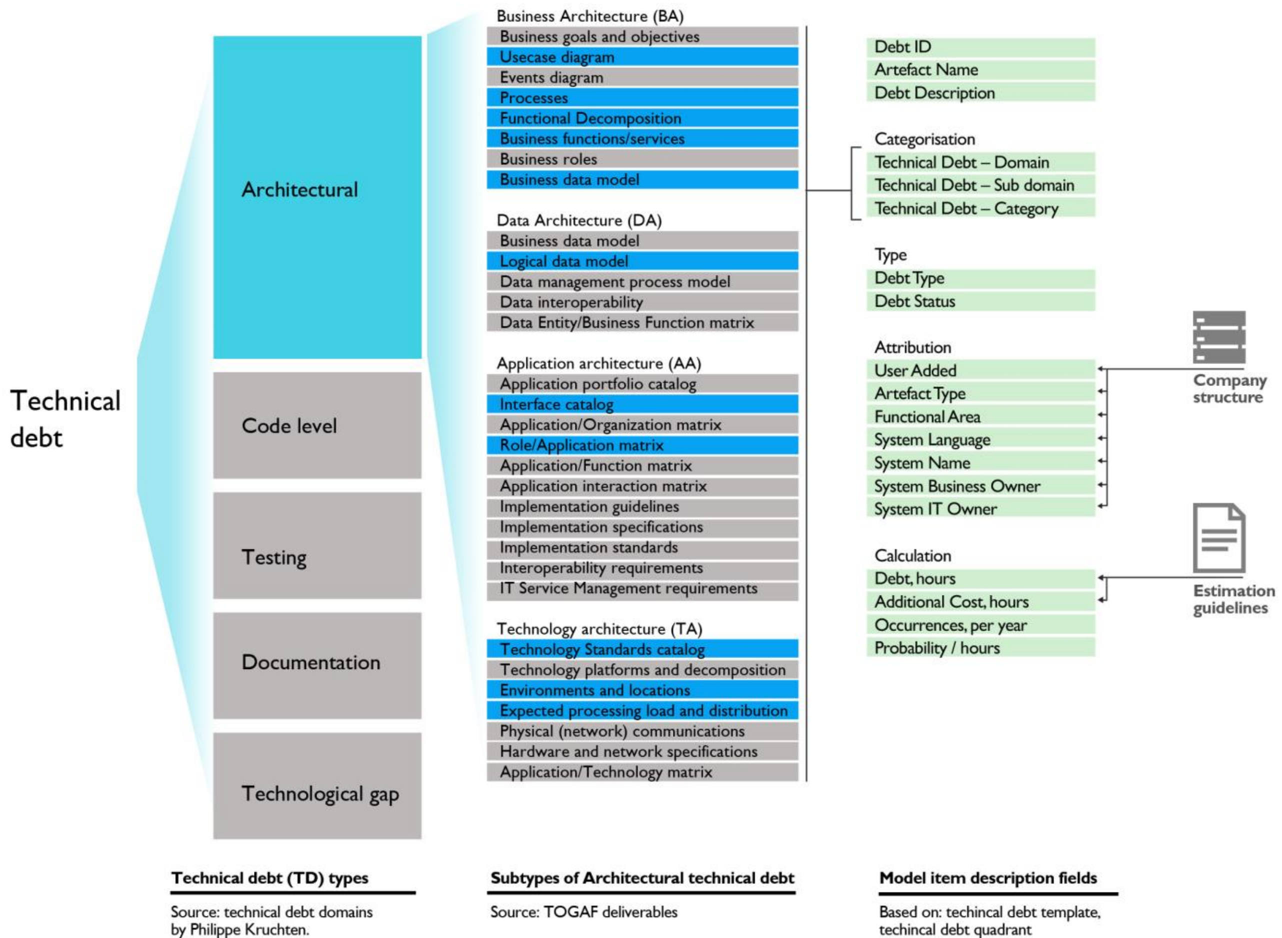


Figure 13. General model view

4.2 Model parts

We will describe the various parts of the model, starting on the left-hand side.

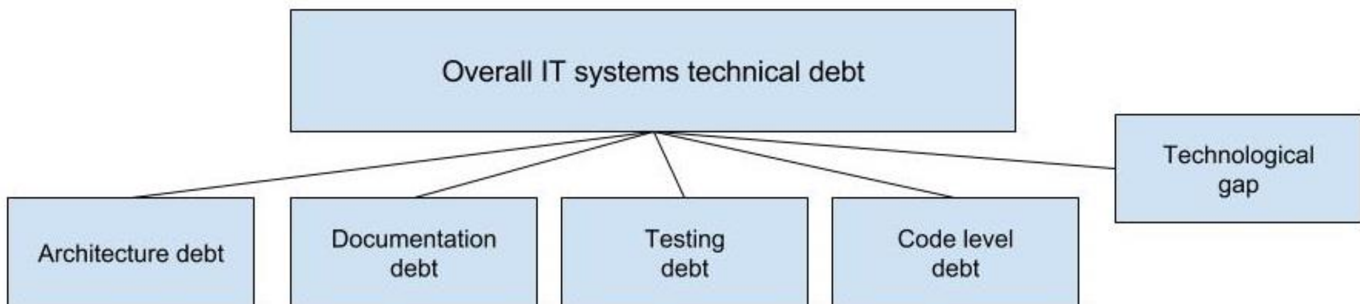
4.2.1 TD taxonomy

Developed taxonomy (Fig. 3) presents all technical debt categories defined in current model version. The five major debt types from Philippe Kruchten TD model are used in [26]. Each block contains terms and a definitions list that were intended to provide common language for discussion for project stakeholders:

- Researchers working on overall TD identification and measurement
- Practitioners, specialists in one of the investigated areas (testers, developers, architects etc.)
- Product owners and business representatives

Each TD category is supplemented with additional point *domain-specific types*. This is planned to be an extendable item that will contain metrics that can be valuable for specific domain that company operates in. List of domain-specific types and its impact is be defined though cooperation with local practitioners.
Full schema of TD taxonomy developed is presented in Appendix 1.

Figure 14. TD categories taxonomy



Information on definitions was collected referring to acknowledged scientific papers (IEEE, BSC) as well as industrial standards and approaches (ISO, TOFAG, SWEBOK).

Architectural debt

Supporting methodology for defining ATD categories was chosen according to a EA framework which was used in the company – which is TOGAF. So as subcategories we use "outputs" listings that are described in TOGAF for four layers of system architecture: Business architecture, Data architecture, Application architecture and Technology architecture.
4 subcategories of ATD are decomposed into following list of subjects according to TOGAF project implementation guidelines. Applicability of each of them is still a question that needs to be investigated. Later stakeholders can estimate amount of TD per subtype.
We used TOGAF as a backbone because of several factors:
It's used in ABN AMRO internally for several years so it is a common language for company's employee stakeholders – architects, project managers, developers, testers, etc. TOGAF is a basement of ABN AMRO application reference document, which provides a model architecture approaches for structuring applications and providing principles, policies and standards that govern its components.

TOGAF framework is acknowledged by professional community as a high quality product. The framework was initially developed in 1980's US Department of Defense and was called TAFIM[38]. Later OpenGroup[39] reworked it partly and launched TOGAF 1.0 in 1995. Since that date it's still being reworked and improved constantly. The latest version is TOGAF 9.1 that was released on 1 December 2011. All the framework descriptions and schemas are open source which makes it possible for any company to apply it for its architecture without incurring significant licensing costs.

Documentation and testing

Initially types and definition from ISO/IEC 26515:2011 and SWEBOK were used. Later the list was adjusted according to techniques list and problems employees (testers, architects, developers) identified in specific domains during preliminary interviews.

Code level

It was planned that Sonar will be used as ASA tool for collecting data con code-level debt. According to the model, results on code level TD will be filled in by metrics obtained from Sonar tool. This will add practical value considering one of the targets of the project to identify spots with maximum TD load.

However at during implementation of Sonar on one of the models – smaller application in IB team experienced technical difficulties. So for now results on Sonar analysis of codebase of IB systems is not included into the scope of the current thesis paper.

Technological gap

Collection of artifacts that involve usage of outdated libraries applications and frameworks. This category of debt is different from others by its nature. Other categories basically present listing of features that are involved in to software systems development and maintenance accompanies with their status (documents absence, poor development, good state). Estimation guidelines are used to convert status of the item into practical hours-measured value.

While TG items serve as a collection of cases on different layers of the systems architecture that describe outdated technologies that are used.

In current state of the model we outlined several than subtypes of technological gap items. Some of them are the following:

1. Server side: Java codebase issues
2. Server side: Application server version issues
3. Client side: Javascript/CSS codebase issues

The relative impact of each of technological gap categories is still a question for further investigation.

There is awareness present in software development community about the existence of such a factor as passive aging of systems. It means that systems that were built according to latest s approaches at one time can become outdated and incur sufficient amount of debt several years later. Methods to measure this effect numerically and provide effective reaction are still to be investigated.

[38] Web link: http://pubs.opengroup.org/architecture/togaf8-doc/arch/chap37.html#tag_38_11

[39] The Open Group is a global consortium more with than 400 member organizations, that enables the achievement of business objectives through IT standards. Web link : http://www.opengroup.org/aboutus

4.2.2 Item description

Our research into how should item description look like we started from technical debt template [7]. We have enhanced it with fields that will contain specific details that would be needed for future analysis of the results in context of project requirements. For example the Meta data field - that will contain all the information about item localization in company's infrastructure. As a result we obtained the following group of parameters that are presented in Table 1.

Table 6. Extended TD measurement template

#	Name	Description	Source	Units
1	TD principal	Contains TD principal measurement	Practitioner feedback	hours
2	TD additional cost	Contains TD interest measurement	Practitioner feedback	hours
3	Occurrences	Number of cases when code was edited per year, actually payments on captured TD	Documents analysis	number
4	Probability	Represents amount of debt per year that is unexpectedly paid	Multiplication of 2) and 3)	hours
5	Generic Debt Type	Derived from TD quadrant	Expertise	typed
6	Generic Debt Status	Describes Accepted	Expertise	typed
7	Meta data	Field containing information about date added, author, systems owner, functional area, language	Expertise	text

Further on, considering details of technical implementation we have reworked the extended template and added fields representing taxonomy categories as well as more meta data fields. In a final Microsoft Excel table that was used for data gathering we had 19 columns. Category titles and groups description is presented in Table 2.

Table 7. Item description fields in model used

Fields	Description
1. Debt ID 2. Artefact Name* 3. Debt Description* 4. Artefact Type* 5. Functional Area*	General information – system name and its functional categorisation.
6. Technical Debt – Domain 7. Technical Debt – Sub domain 8. Technical Debt - Category	Debt item categorisation – according to used TD taxonomy
9. User Added* 10. Debt Type 11. Debt Status 12. System Language*	Debt type and categorisation; item reference in ABN AMRO systems structure – both on technical and business level.

13. System Name* 14. System Business Owner* 15. System IT Owner*	
16. Debt (hours) 17. Additional Cost (hours) 18. Occurrences (per year) 19. Probability /(hours)	Details of debt calculation – digital parameters of captured debt in the artefacts

Fields marked with asterisk (*) were removed or anonymised due to security regulations in ABN AMRO. Codes used in those fields are random and do not relate to any internal names of components.

4.2.3 Estimation guidelines

For each debt category definitions of deliverables are derived from TOGAF standard definitions (Appendix 2). Each deliverable in TOGAF methodology serves as a summary of the development step activity. For future usage it helps to develop new systems with reducing duplication and other negative effects of bad architecture. It also provides base for next step of architecture development.
For each category we have 3 types of activities: creating, updating and modifying. Each activity has different estimated time due to difference in activities and expertise needed to perform this action.
For example creating a document takes most time because it involves the such activities as – observing the existing systems profiles, investigating associated artifacts and communication with stakeholders about implementation details, functionality, etc. and formalizing the gathered information in a form of a deliverable.

Estimation guidelines can be called one of the central concepts of the developed method. Because they are a proving means to convert perceived project status into measurable amount in hours. One of the main tracks for future method improvement is to them unambitious and also probably define commonly happening occasions in the architecture and define how a participant should act in each case.
Estimation guidelines we're based on the expertise of practitioners. Some of the Practitioners that are working on the IB projects were also the participants in method evaluation survey (part 6.2.2). Considering their education level, professional experience and experience with system utilized in ABN AMRO especially we concluded that this can serve a sufficient foundation for first data collection round.

4.2.4 Calculating and aggregating tool

Reasoning about available options for technical implementation possibilities we considered the following options:

1. Microsoft Office Excel sheet with validation rules and pivot tables;
2. Questionnaire or survey (paper or web-based);
3. Specific TD dashboard web application.

Due to time limitations and possible current stage of the project we have chosen option 1, implemented in current research:

4.3 Model discussion

Communication with practitioners at ABN AMRO

We had several meetings with local practitioners in different domains: testing dept. representative, developers, architects. Meeting consisted of initial project presentation, explanation of the term and its structure and influence on overall process. Later interviewees were asked to give feedback. Feedback included 2 types of information: details about specific procedures used in the company and identifying directly some cases where it was obviously occurrences of technical debt.

Table 8. Examples of TD issues at ABN AMRO

TD domain	Issue
Technological gap items	Usage of outdated Java version Usage of outdated WebSphere version Usage of outdated STRUTS version Usage of outdated front end library
Testing domain	Insufficient amount of unit tests Lack of automated test
Architecture level debt	Localized tables with migration jobs Not used business configuration objects

Chapter 5. Application of the model

This chapter describes the case study data collected during the research. First it goes through theoretical findings and later presents data on systems that was discovered.

5.1 Introduction

Planned research structure included the following steps:

Model development and adjustments

This stage included constructing the model, communication with practitioners to adjust the debt taxonomy. We used TD template as a basement and GQM method to refine the model fields.

Technical implementation and data collection

The model was implemented as a file with data structure (debt taxonomy, TD item fields) and validation to collect measurements. We also defined the estimation guidelines that were used by participants to estimate the amount of debt incurred in each case. Employees were filling in the model file for system case by hand. As is was the first attempt to collect data we selected 5 systems and 3 employees for data collection.

Results analysis and feedback questionnaire

After data is collected analysis of data was carried on. This will included creating a pivot table to present TD distribution across systems, debt types and architectural layers. This also helped us to identify the spots with highest amount of TD interest and principal. Later the questionnaire was sent to data collection participants to measure the feasibility of the model.

Below each of the research steps is described with results obtained.

5.2 GQM results

After several iterations the following goals were formulated. Then they were connected with appropriate questions while those in turn formed connection to metric used in the model.

Goals

1. Identify are types of TD we should measure
2. Save costs on most often changed components
3. Identify owners of TD issues across systems
4. Identify domains with debt incurred by time of creation / old architecture/code

Questions

1. What EA methodology is used?
2. What is amount of hours of debt associated with each component
3. How often they are updated?
4. Who is the owner of the component?
5. What part of documentation is missing?
6. How often is documentation for modules/components is updated?
7. What are documentation types used for projects?
8. What automated testing tools do we use?
9. How are organized types of testing that are used?
10. Who is in charge for creating test cases?
11. How big is duplication of code in current code base?

12. How many lines of code is our average class/method? What are the longest classes/methods?
13. Do we rely on outdated frameworks, libraries, applications?

Metrics

1. Component description (meta-data)
2. Issue description using TD template (meta-data)
3. Group of metrics: architecture and subcategories (TOGAF + prioritization)
4. Group of metrics: code (SQALE + security)
5. Group of metrics: documentation (according to documentation types used)
6. Group of metrics: testing (according to test layers used + automation)
7. List of technological gap items

In graphical representation GQM results are presented on Figure 14.

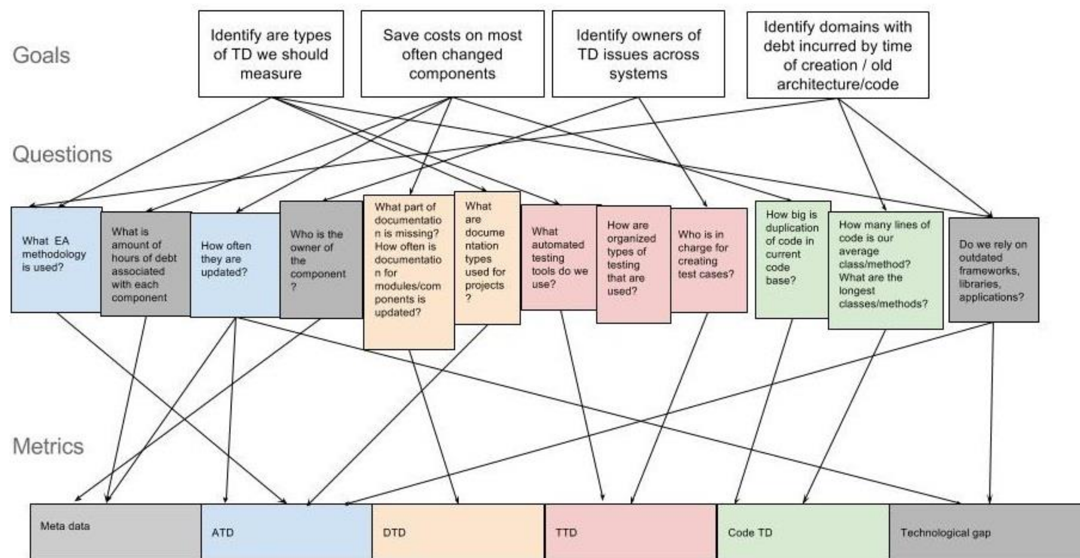


Figure 15. Goal question metric results

5.3 Technical implementation

Currently model measurement tool implementation done as a Microsoft Office Excel sheet with pivots table. This gives fast and easy tool for both collecting and presenting data. Later model can be re-written as a web-application.

Manual for the subjects participating in technical debt data collection

It's expected that whole process won't take you more than 20-30 minutes.

1. Go through the presentation of the project (attached) – to get the idea of TD terminology. Check that you clearly see the difference between TD principal and TD interest.
2. Open model Excel file (attached)
    a. Add the system using its code (e.g. "XXX108") that you're going to measure as a new row if it's not present in the list yet.
    b. Fill in the columns in meta-data section with data related to the system.

c. Go through the sub-categories of architectural, test or documentation debt and fill in the cells in that row. Definitions of categories are in the attached reference document.
d. For each category, please add descriptive categorization in terms of was this debt taken by occasion (reckless) or by intent (inadvertent).
e. Estimating amount of debt can be done using following scale: fixing the issue or going on without fixing it will take you or your colleagues: hours/days/weeks/months.

Participants of the measurement

Participants of the questionnaire were the same practitioners who added data about technical debt amount during the main phase of the research. As it can be seen in table below they all are experienced professionals in the architecture and development area.

Table 9. Employees and systems measured

Employees / Systems	XXX040	XXX001	XXX509	XXX245	XXX001WS	JAVA BATCH
PA2778			X			
JI0108	X	X			X	
WA2126	X	X		X		X

List of systems

Systems researched were built during in house projects run by 3rd party contracting companies (IBM, TCS). Investigated systems present different layers of the IB infrastructure. They all are written in Java (JEE)as main programming language. They also utilize SQL for some parts. Ages of those systems vary from 8.5 to 2 years. Mode details about those systems are summarized in Table 10.

Though around 10-12 of business applications used by IB are planned to be retired in the next 2-3 years, none of the investigated systems fits this category. Total size of all IB systems is approximately 1881 KLOC.

Table 10. Researched systems characteristics

Systems	System size, LOC	System age, years
XXX001	26668	2.5
XXX 040	2003	8
XXX 001WS	61788	4
XXX245	7656	2
XXX 509	17569	3

5.4 Data collected on technical debt

Results table (Table 10) describes summary on TD for 5 researched systems. All of them act on different layers of IT infrastructure in IB systems at ABN AMRO. The detailed TD model filled in is presented in

Appendix 5. Data collected was only related to architectural TD categories. This presents the focus of the current paper with is architectural technical debt estimation. It also presents smaller amount of data itself but we find it sufficient enough to proceed with initial model evaluation. Another reason was that estimation guidelines for other types of debt (Testing, Documentation) are still in discussion and formalization phase.
As it was described in the TD taxonomy we have selected 11 subcategories for ATD. While in the final table one can see smaller amount of subcategories (4 to 7) in each systems description. This is because 5 studied systems present different functional layers. As TOGAF categories describe different levels of EA the not all the 11 ATD subcategories can be applied to each system. If we consider all the lines of data collected (see Appendix 5), not filtering for specific system – all of the 11 ATD subcategories are present there.

Table 10. Summary on collected TD data

System	Debt principal	Change occurrence per year, avg.	Debt interest per year	Debt categories described
XXX001	84	4	57,8	4
XXX040	22,4	1	5,8	6
XXX001WS	37,5	2	11,25	4
XXX245	11,9	2	11,3	4
XXX509	188,9	2	108,3	7
JAVA BATCH	40	3	18	1

General notes

All architectural TD is considered to be "Prudent" type according to TD quadrant. This is was done because we came to the conclusion that on system architecture level any decision is motivated and stakeholders are aware of it. Also all debt data was described as "Just identified" in "Debt Status" column. This is because collected data was the first set obtained and it was reasonable to mark it as "Just identified". As it is expected later TD data collection model can become a part of regular metrics and then this field can contain more detailed information on project TD evolution.

Results for researched systems

The most TD principal and interest incurred is by system XXX509, having 188,9 and 108,3 hours respectively. It can be seen that this item contains the most ATD categories involved into the description – 7. This fact could influence the amount of calculated debt for this system since the model calculations was not really adjusted before and this specific system was not cross-measured by different employees.

While the lowest rate is at XXX001WS system, which is 11,9 and 11,3 hours respectively. Also the date of initial release of the system can have a big impact on the amount of registered architectural debt – for relatively older systems there can be better state of the required architectural documentation provided than it is for newly developed ones.

If we consider the ratio between the TD principal and interest incurred by different systems we can see that the largest factor will be (i.e. worst in terms of) in case of XXX245, it's close to 1. While the amounts of

debt are almost the smallest in absolute numbers, this means that in relative amounts, debt of this system is the most expensive – *yearly a interest payments* are 97% (11.3/11.9 x 100%). Such types of debt in systems should be eliminated in first place in case if this system is not planned to be retired in reasonable time.
If we compare those results with XXX040 for which this metric has the lowest value of 0.26. One of the reasons can be that the number of change occurrence per year is lowest, estimated as 1. This case presents a type of debt that should be identified as such and tracked but can be left intact due to its low costs.

As the XXX001 system has the greatest number of change occurrence per year, which is 4. This lead to the fact that having quite small amounts of additional costs per category lead to high amount of yearly interest.

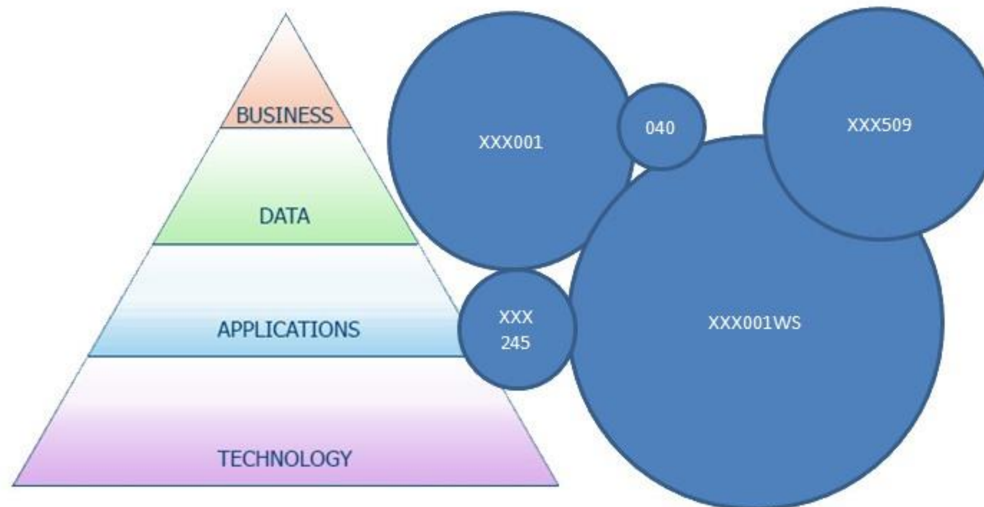


Figure 16. Researched systems compared

Analyzing properties of the systems, such as their attribution according to TOGAF layers or systems size or the amount of debt incurred we could not derive any more direct regularities. One of the reasons for that could be relatively small amount of data collected.
On Figure 16 is presented a diagram showing comparable size of the systems and their technical debt items attribution, referencing TOGAF methodology categories.
Here we would like to recap our conclusions for investigated systems, considering their technical parameters and measured TD amounts for them:

- System XXX040 is a smallest by its codebase and is probably modular – it's changed approximately once in a year, which is good. While the system itself is relatively old – 8 years. This occasion should not be treated as urgent TD that must be eliminated.
- XXX245 is the system that has the biggest TD percentage rate, and is relatively small, and new, so this inconsistency should be probably fixed first.
- For systems XXX001WS, XXX001 that have the biggest sizes in the measured group and also are second in size of TD percentage rate. Those points should be fixed in second turn.
- XXX509 has average amount in all parameters but it has biggest amount of absolute debt in hours.

Comparing 2 employees answers about the same systems

For 2 systems (XXX001 and XXX040) we had 2 different employees adding in data into the model. This intersection helped us to compare the results and evaluate model feasibility. Considering our amount of data collected we had only 2 differences in results provided by different participants.

1) When estimating TD data for system XXX001 participants estimated debt associated with Use Case diagram differently in 3 times – 72 hours against 24.

The reason behind is that one of the participants had a wrong approach to estimating amount of use cases for each business function. In this case there were 3 separate blocks with similar functionality, and there had to be one use case for all 3, but not a separate use case for each. That's how 3 times difference emerged. Referring back to the method we used, such inconsistency could be fixed in future by having more detailed description in estimation guidelines.

2) In XXX040 case we had different participants choosing different categories to describe incurred debt. While one respondent selected Logical Data Model, another used Process Flow diagram and Role / System Matrix.

In this case the participants approach was different because of difference in their perception of what deliverable should have been used was different. Talking about the method, it could be improved to avoid such collisions next time to by providing pre-filled architectural categories to the participants - to strictly define what participants should measure.

Virtual components

While collecting data we came to understanding that model can also contain high-level "virtual" components. Such an approach can provide a way to capture TD in architecture on different levels while the system or a module is not physically one piece of software. This was done introducing "Java batch" system in the list of researched systems. Is not a real system but a virtual aggregation of system modules and components. It can be referenced as a backend platform providing services needed for normal functioning of front office systems.

Other notes

During data collection we had another example of unreasonable widening the technical debt metaphor. According to internal regulations at ABN AMRO all the technical documents are to be written in English. But one of the participants while examining available information discovered that some of them related to one of the systems are written in Dutch. For him being a non-native Dutch speaker it will take reasonably longer to appropriately identify current document status and propose possibly existing debt. So he asked whether this should be called a debt of some type. Sharing the position of Robert Martin[40] towards bad code, this can be qualified as violation of adopted work practice (compare with badly written code in code level metrics for example) that just needs to be fixed but not a specific type of debt captured by system.

---

[40] Blog post, 09/22/2009, Web link: https://sites.google.com/site/unclebobconsultingllc/a-mess-is-not-a-technical-debt

5.5 Evaluation of proposed method

For evaluating the model we used additional feedback questionnaire for all employees who were participating in data collections on architectural technical debt. More details on the participants employees is presented in 5.2. Evaluation questionnaire consisted of 3 logical parts :

1. Question about employee's experience in the field and in company projects
2. Questions to measure model feasibility
3. Comments of data collection process and general feedback

Full listing of results in presented in the Appendix 5 and 6. Below results discussion is presented.

Model feasibility measurement

For discussing model feasibility we had the following questions list:

4. If the research question about is technical debt measurement, to what degree do you think using the proposed model will enhance your job performance?
5. To what degree do you think using this model for estimating TD would be easy to use?[41]
6. To what degree do you think this model is consistent with the existing values, needs and experience related to TD awareness ?
7. If the research question is about technical debt measurement, to what degree do you think you would use this model in future?
8. If the research question is about technical debt measurement, to what degree do you think other colleagues of yours would use this model?
9. To what degree the model is able to measure the amount of TD incurred by system?

For each of the questions respondent could give one of the following answers: Very low degree (1), Low degree (2) , Average (3) , High degree (4) , Very High degree (5) . The results collected are presented on Figure 4.

[41] Question text in Q5 was reformulated to fit other questions approach. Original version was: To what degree do you think using this model for estimating TD would require much effort to use?

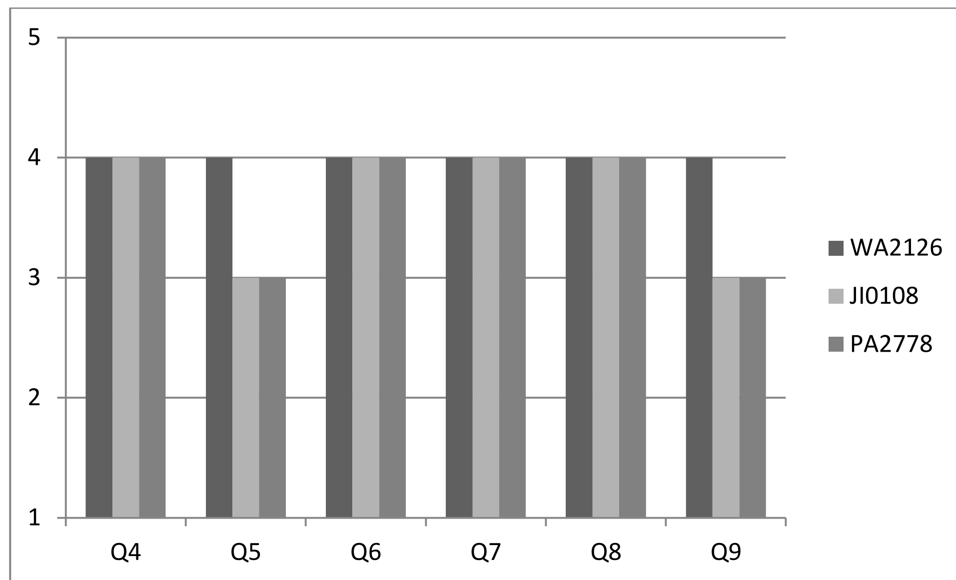


Figure 17. Feedback questionnaire results

As it is presented on Figure 15, mostly participants answers rated the method they used quite high.
The model is stated to be especially good in sense of expected fulfilling the requirements on collecting data on TD in the systems (every respondent rated it 4 out of 5). It also can be easily speeded across their colleagues (every respondent rated it 4 out of 5).

The lowest perceptive mark (3, 3 and 4 out of 5) was graded the ability of the model to capture all the TD incurred by systems. We see this as a result of the fact the model still does not include several parts describing other valuable domains of TD in the system.

Usage of a tool also implies understanding of its principles, so we wanted to know about difficulties participants had. That was done by next question - Q11: *Did you have any difficulties while understanding the model or filling in the measurement table?* The following responses were collected:

- Not much;
- I had to refer to the category definitions in repository. I was able to understand it with detail text
- Not really

As we can conclude from given answers, that for employees that are experienced enough with systems that are measured model and the task was well understandable. However interaction with model during adding the data may involve additional activities for searching and obtaining needed reference materials.

Comments on data collection process

Our questions to get feedback on what's missing in the model was: Q10: *Do you think there are aspects that are missing in this model?* The following responses were collected:

- There should be some way of capturing the relationship between different categories. E.g; an incorrect Use case diagram (business architecture) can influence technical debt in testing;
- We only calculated for architectural debt. Will be more clear if others are also done.

Participants replies collected on abovementioned question directly support the point that we discovered during initial stages of our research. This issue with inner relationships between different categories or possible duplication was also mentioned by other practitioners in the company before. On current stage we have no clear solution how this influence can be eliminated for data collected. We see two possible ways to solve such type of inconsistency on further stages of model development:

- As on empirical level, the elimination can be implemented as a set of rules – that would be a part of the model method, the same as estimation guidelines. Such rules could be directly stating which types of debt should be measured in what categories, with complete ignoring of the same issue in others.
- Considering TD amounts calculation, the registration of existing interrelations could be done by introducing specific coefficients that must be added to model calculation formulas.

Both approaches however require significantly more volume of data collected from various systems.

As the model in the future could be used as a method to measure the TD status of other large information systems in the company and would involve more participants we wanted to estimate how much time it takes to work with it on a current stage. The question we had was - Q12: *Please estimate how much time it took you to fill in the measurement table (in total or per component)?*

Findings on that questions differ for each of the participants, quite noticeably. Those data is presented in Table 10. For better presenting our point we have added also the details about the experience of each of the participants in the area and specifically within company projects.

Table 111. Comparing time for filling in the model with data

Question / Employee	WA2126	JI0108	PA2778
How long have you been working in ICT sector?	13	7	6
How long have you been working at ABN AMRO projects?	8	7	3
Please estimate how much time it took you to fill in the measurement table (in total or per component)	0.5	1	2

As it can be see seen in the Table 10 there is a direct relationship between years of experience of the employee and the time needed to fill in the data about the systems. Even considering considerably small dataset we have now this tendency supports common sense expectations on the questions.

As it was discussed in 6.2.1, we have compared results collected from different participants about the same system and found that they differ. Also to check the model consistency, we have asked the following question in our feedback questionnaire: *Q13: Please give your comments on following results of comparing data collection: There are 50% changes in list of TD categories describing XXX040 component.* We got the following answers:

- Employees have personal different professional experience
- Employees have personal view on TD amount estimation

We think that those replies also support the results of the previous question. Experience of the employee plays a significant role in correct estimation of the system's issues and possible amounts of incurred debt. To eliminate "personal view on TD amount" the estimation guidelines as well as the other empirical parts of the method must be adjusted and made more formal.

5.6 Threats to validity

During the model development we came across several points that should be considered as things to be solved in future work.

Duplication on functional level is not measured

On code-level metrics duplication of code blocks occurrences is defined as one of the factors leading to worse code quality hence technical debt. For code level, however such occasions can be measured and evaluated automatically.
On architectural level there also can be the case of 2 or more systems or system parts having similar functionalities up to some degree. Such cases can be common for cases with wide and diverse IT systems in companies. But investigation of such occurrence of duplication in each case has to be investigated in person. The currently developed model does not include any approaches towards such duplication evaluation.
One of the ways to handle this could be interviewing stakeholders from different functional domains. Also detailed analysis of all existing components catalogues/portfolios can help to enumerate such issues and measure occurrences.

Only ATD domain data collected

In current research only architectural types of technical debt were investigated. Obtaining additional set of results by investigating the same systems from other angles – code and testing debt, could give a more detailed picture on the general debt load of the those systems.
This was not discovered because of 2 reasons: first, we could not include this because of the time limitations of the project. Second comparing those results would require much more work on a model level – how one could properly compare such different pointers.

Estimation of amount of hours

As it was stated earlier, estimation guidelines are a very valuable part of the developed method. They directly influence the number of registered debt in each of the categories. As such change of expected amount of hours to fix in estimation guidelines can change the results collected.
We propose that estimation guidelines that need to be adjusted every time when the new larger data set will be collected.

Appropriate systems selection

In the IB domain there are up to 150 systems and modules present. For our measurement we have selected 5 systems. This selection is based on professional experience of the participated researchers and their expectations that those systems expected to have biggest debt rates.

Their long time expertise in company's systems and diversity of selected systems (on architectural levels, which is discussed in 5.2) makes us to conclude that selection was done in proper way considering research setting.

Correct ATD categories selection

For our measurement we have selected 11 ATD categories of 34 that are present in the TOGAF standard for this stage. This selection is based fully on professional experience of the participated researchers.
This choice had to be done due to setting of the project, but we agree that this can lead to underestimation of the total debt presence in the researched systems. Question of what should be chosen for estimation - another subset of categories or full list of ATD categories is still open. The ways to solve it require more data to analyze and possible more professional feedback expertise.

Too little participants during data collection

During data collection we had only 3 employees as participants for data collection on debt amount estimation. We have to agree that larger number of participants could give us better data to discuss validity and feasibility of the model and its parts. Unfortunately this was dictated by the setting of the project on its current level of development. In future to make the method more mature it will be needed to implement a larger scale of data collection to be able to apply not only logical reasoning but also statistical data analysis to see other drawbacks in the model.

Correct participants selection

On data collection stage all the participants were representatives of one 3rd party contracting organization that worked with ABN AMRO project for several years already.
There can be an argument that the chosen ones are too closely related and may have close point of view on the details of the systems architecture. Considering their personal experience and experience with company's projects (Table 10) makes us think that this is mature enough to have reliable results in this measurement as separate respondents.

Chapter 6. Conclusions

In this chapter we summarize the findings of the research and provide answers to the research questions. Additionally, we discuss method limitations and later indicate possible areas for future work.

6.1 Summary

First we answer the research questions formulated in the beginning of the paper.

RQ1: What is architectural technical debt, how can it be measured, and how does it related to other TD measurement techniques?

In this research we have presented a definition of architectural technical debt in the form of a taxonomy, based on literature including TOGAF. Further we have defined the method including item template and estimation guidelines for measuring technical debt and recording technical debt items in a structured manner. In our comprehension of architectural technical debt we follow Philippe Kruchten's approach in positioning it as complementary domain to four other main types of technical debt.

RQ2: How can ATD best be measured in enterprise IT systems such as those of ABN AMRO?

It was presented that the measurement model that we developed can be applied to the enterprise systems of ABN AMRO by providing guidelines and measurement tools for the practitioners that are familiar with the systems. By this we can state that model described in current paper can be one of the comprehensive methods for investigating TD in enterprise IT systems.

RQ3: How feasible, useful and reliable are the proposed measurements of ATD in practice?

1. Feasibility was shown by applying the model to 5 systems representing different parts of the company's IB applications stack
2. Usability was shown on the basis of the perception of practitioners. Practitioners find the proposed model and the approach easily understandable, and compatible with their professional industry experience.
3. Reliability was confirmed by comparing the results of measurements of the same system by different participants.

However initially the method itself should be assessed and adjusted based on wider data set. This will help to avoid subjectivity of measurements, lack of coverage for specific details and possible mismatches.

6.2 Discussion

Summing up the results of the evaluation we can highlight several points:

1. Model structure and concepts on which it is based are logical and easy to understand for practitioners;

2. Model has a good relation to practitioners area of expertise and contains references to the architecture concepts that are used by practitioners in everyday work;
3. There are some occurrences of possible ambiguity in estimation guidelines that can lead to different estimation results by different practitioners;
4. TD estimation process participants should have a considerable experience with the systems they are going to measure, otherwise the results can be much less reliable;
5. Currently developed model covers only architectural part of technical debt. Better feasibility discussion can be done when more data on other layers will be collected.

6.3 Contributions

Summary

We have started from investigating the theoretical state of the art approaches using literature sources. Then we developed the model that can be used to capture the technical debt occurrences in the IT systems of the company.

We started from describing overall technical debt types. Then we added more detailed categorization of each debt type. For more deep investigation the domain of architectural debt was chosen. TOGAF was used as a backbone methodology for providing ATD categorization. When the practical tool was outlined we have selected several most valuable systems for initial investigation.

Data about TD incurred for each system was collected using the TD model, where practitioners could fill in the details about estimated amount of debt interest and principal for each debt category for the system. In total we have collected data about architectural technical debt in 5 IB systems.

To evaluate the developed approach we've conducted a stakeholder's survey to collect their attitude to the model they had to use. In this survey we asked them to measure the usefulness of the model for them, their projects in future and their colleagues. The results of the evaluation are described in block 5.4.

The results obtained can be used as a basis for performing a refinement of the method - developing a more usable tool and dashboard, adjusting estimation guidelines, providing additional methodological recommendations. After that the method can be adjusted and applied for the other systems in the IB environment to collect and analyze this information.

Technical debt management proceedings

Our research is an initial step in implementing the TD management method in ABN AMRO. Theoretical approaches on TD management were described in Chapter 2.5. Considering the practical steps we see the following principles that should be implemented:

1. Define debt interest threshold. So that after assessment, the management could have a clear picture which cases should be the first targets for rework.

2. Visualization of the data collected is one of the means to give the topic more attention
3. Describe managerial mechanisms to establish a feedback loop between data on technical debt collected and actions that must be performed for most scored systems.
4. Implement a plan do check act approach - when measurements are done on a regular basis. Also the calculation formula has to be reviewed regularly based on the results obtained.

Feasibility of the method

Considering all of the above we conclude that the proposed method can be a starting point for developing IT debt management methodology. However current state should be improved in many aspects (see limitations part) to become a really practical methodology.

First, much more data needs to be collected to see other effects and misconceptions of the method. A larger number of participants and wider range of investigated systems can provide data for statistical analysis that can highlight drawbacks of the methodology.

Special attention should be put into estimation guidelines because if their extreme value for obtaining numerical data on technical debt in the system.

6.4 Limitations

In the subsection, we discuss the method limitations we identified during our research.

Model completeness

During this research the model for identifying different types of TD was developed. It was based on several theoretical approaches derived from scientific literature. This model follows requirements of the practical TD investigation project at ABN AMRO. It also implements some of the state of the art approaches of current technical debt landscape. But considering regular evolution of the TD topic in last year's we and specific needs/requirements that can be in other organizations we must state that some of the aspects are not covered by it.

Subjectivity of measurements

Several key points of the research (estimation guidelines, data on TD items itself) are based on expert opinions of participants of measurements and project team which can be subjective. Estimations done by participants are highly dependent on their professional experience and on familiarity with the systems researched. Answers given by the consultants are highly dependent. Nevertheless we implemented evaluation feedback questionnaire and also cross-measurement of one system by two different participants. This was done to evaluate model feasibility and to provide input for further improvements of the model.

Model extendibility

Considering a question whether the model developed can be applied outside the context of current case studies. We think that for systems comparable by size and types the model can be a good start in reasoning for other practitioners while building their own TD capturing

strategy. Direct model applying without adjustments in list of debt types/categories and methodology used will not give the correct results.
As we got the sufficient feedback from participants involved in the research this makes us conclude that adoption level of the model in future can be expected to be good. Valuable point is that participants must have a sufficient level of knowledge in both system they're planning to measure and technical topic of EA and software quality assessment. If we consider expanding the method to apply for the whole company IT systems landscape our point is that - there can be no one person/team who can measure all TD across all systems or departments. Method application must be done by each group of stakeholders separately based on common principles that are defining the model and estimation guidelines.

Usage of reliable instruments

This research utilized two instruments. First, goal question metric approach, to validate the model defined based on reviewing literature on the topic. This retrofitting analysis presented good results. Second, we added feedback questionnaire to see the perceived usefulness and reliability of the method for the future development. This part also presented good results.

6.5 Future work

Following points should be first subjects for future method enhancement:

1. Broadening method application on other systems within company's IT landscape as well as on other software modules. This will lead to collecting more data that can provide better reasoning to see additional improvements that are need to be done in the model. It also means involvement of more practitioners into the measurement activities, this will also help to collect more professional feedback and additionally improve the method.
2. Possible duplication of incurred debt in different categories. Research work need to be done to figure out how this can be eliminated or at least reduced to an adequate level.
3. Improvement and broadening of estimation guidelines. This part of the method must be researched against different layers of TD (testing, documentation, etc.) and also proposed amounts of time could be corrected considering more data to analyse and additional feedback from practitioners.

Literature list

1 Distributed Agile, Agile Testing, and Technical Debt
Raja Bavani, 2012 IEEE Software

2 Estimating the Size, Cost, and Types of Technical Debt
Bill Curtis, Jay Sappidi, Alexandra Szynkarski, MTD 2012, Zurich, Switzerland

3 Towards a Model for Optimizing Technical Debt in Software Products
Narayan Ramasubbu, Chris F. Kemerer; MTD 2013, San Francisco, CA, USA

4 In search of metric for managing architectural technical debt
Robert L. Nord, Ipek Ozkaya, Philippe Kruchten, Marco Gonzalez-Rojas, 2012 IEEE CS

5 Tracking Technical Debt — An Exploratory Case Study
Yuepu Guo, Carolyn Seaman, Rebeka Gomes, Antonio Cavalcanti et oth., 2011 27th IEEE International Conference on Software Maintenance (ICSM)

6 DebtFlag: Technical Debt Management with a Development Environment Integrated Tool
Johannes Holvitie, Ville Leppanen, MTD 2013, San Francisco, CA, USA

7 Measuring and monitoring technical debt
C. Seaman and Y. Guo, Advances in Computers, vol. 82, pp. 25–46, 2011

8 Using automatic static analysis to identify technical debt
Antonio Vetro, ICSE 2012, Zurich, Switzerland ACM Student Research Competition

9 Searching for Build Debt: Experiences Managing Technical Debt at Google
J. David Morgenthaler, Misha Gridnev, Raluca Sauciuc, and Sanjay Bhansali, MTD 2012, Zurich, Switzerland

10 Managing Technical Debt with the SQALE Method
Jean-Louis Letouzey and Michel Ilkiewicz, 2012 IEEE Software

11 Technical Debt as a Meaningful Metaphor for Code Quality
Israel Gat, 2012 IEEE Software

12 Evaluation Criteria Trusted Product Maintainability
Joost Visser, SIG/TÜViT, Version 6.1, Software Improvement Group, 2014

13 The goal question metric approach
Victor R. Basili1 Gianluigi Caldiera1 H. Dieter Rombach2

14 What Is the Value of Your Software?
Jelle de Groot, Ariadi Nugroho, Thomas Back, and Joost Visser, MTD 2012, Zurich, Switzerland

15 The WyCash portfolio management system
W. Cunningham, ACM SIGPLAN OOPS Messenger, vol. 4(2), pp. 29–30, 1993

16 Technical Debt
Web link: http://www.construx.com/10x_Software_Development/Technical_Debt/
2007, Steve McConnell

17 **Minimizing Technical Debt: Developer's Viewpoint**
Vinay Krishna, Dr. Anirban Basu;

18 A Balancing Act: What Software Practitioners Have to Say about Technical Debt
Erin Lim, Nitin Taksande, Carolyn Seaman, 2012, IEEE Software

19 The SQALE Method for Evaluating Technical Debt
Jean-Louis Letouzey, MTD 2012, Zurich, Switzerland

20 Technical Debt
I. Gat, Cutter IT J., 2010

21 Release duration and enterprise agility
Daniel R Greening, IEEE CS, 2012

22 Estimating the Principal of an Application's Technical Debt
Curtis, B.; Sappidi, J.; Szynkarski, A. Nov.-Dec. 2012 v.29 p.34-42, ISSN 0740-7459

23 Understanding the impact of technical debt on the capacity and velocity of team and organizations
Ken Power, MTD 2013, CA USA

24 Practical considerations, challenges and requirements of tool-support for managing technical debt
Davide Falessi, Michele A. Shaw, Forrest Shull et oth., 2013, CA USA

25 Managing Technical debt in software-Reliant Systems
Nanette Brown, yuanfang Cai, Yuepu Guo et oth., 2010, ACM, New Mexico USA

26 Technical Debt: from metaphor to theory and practice
Philippe Kruchten, Robert L.Nord , Ipek Ozkaya., 2012, IEEE Software

27 An Empirical Investigation of Modularity Metrics for Indicating Architectural Technical Debt
Zengyang Li, Peng Liang, Paris Avgeriou, Nicolas Guelfi et oth., 2014, ACM, France

28 Comparing four approaches for technical debt identification
Nico Zazworka, Antonio Vetro, Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman, Forrest Shull. Software Qual J (2014) 22:403–426

29 A portfolio approach to technical debt management
Y. Guo and C. Seaman, Proceeding of the 2nd Workshop on Managing Technical Debt (MTD'11), pp. 31-34, 2011

30 Monitoring code quality and development activity by software maps
J. Bohnet and J. Dollner, in Proceedings of the 2nd Workshop on Managing Technical Debt. ACM, 2011, pp. 9–16.

31 Using Technical Debt Data in Decision Making: Potential Decision Approaches
Carolyn Seaman, Yuepu Guo; Clemente Izurieta; Yuanfang Cai; Nico Zazworka, Forrest Shull; Antonio Vetrò

32 An empirical model of technical debt and interest
A. Nugroho, J. Visser, and T. Kuipers, in Proceeding of the 2nd International Workshop on Managing Technical Debt. ACM, 2011, pp. 1–8

33 An exploration of technical debt
Edith Toma, Aybüke Auruma, Richard Vidgena; The Journal of Systems and Software 86 (2013) 1498– 1516

34 A Threshold Based Approach to Technical Debt
Robert J. Eisenberg, Lockheed-Martin

35 Prioritizing Design Debt Investment Opportunities
Nico Zazworka; Carolyn Seaman; Forrest Shull

36 Measuring Architecture Quality by Structure Plus History Analysis.
Robert Schwanke; Lu Xiao, Yuanfang Cai;
ICSE 2013, San Francisco, CA, USA Software Engineering in Practice

37 Technical Debt from the Stakeholder Perspective
Ted Theodoropoulos; Mark Hofberg, CISA, CRISC; Daniel Kern, PhD
MTD'11, May 23, 2011,

38 Managing Technical Debt in Practice: An Industrial Report,
Clauirton A. Siebra; Graziela S. Tonin, Fabio Q. B. da Silva, Rebeka
ESEM'12, September 19–20

39 A Case Study on Effectively Identifying Technical Debt
Nico Zazworka, Rodrigo O. Spínola, Antonio Vetro et oth.; EASE'13, April 14–16, 2013, Porto de Galinhas, PE, Brasil.

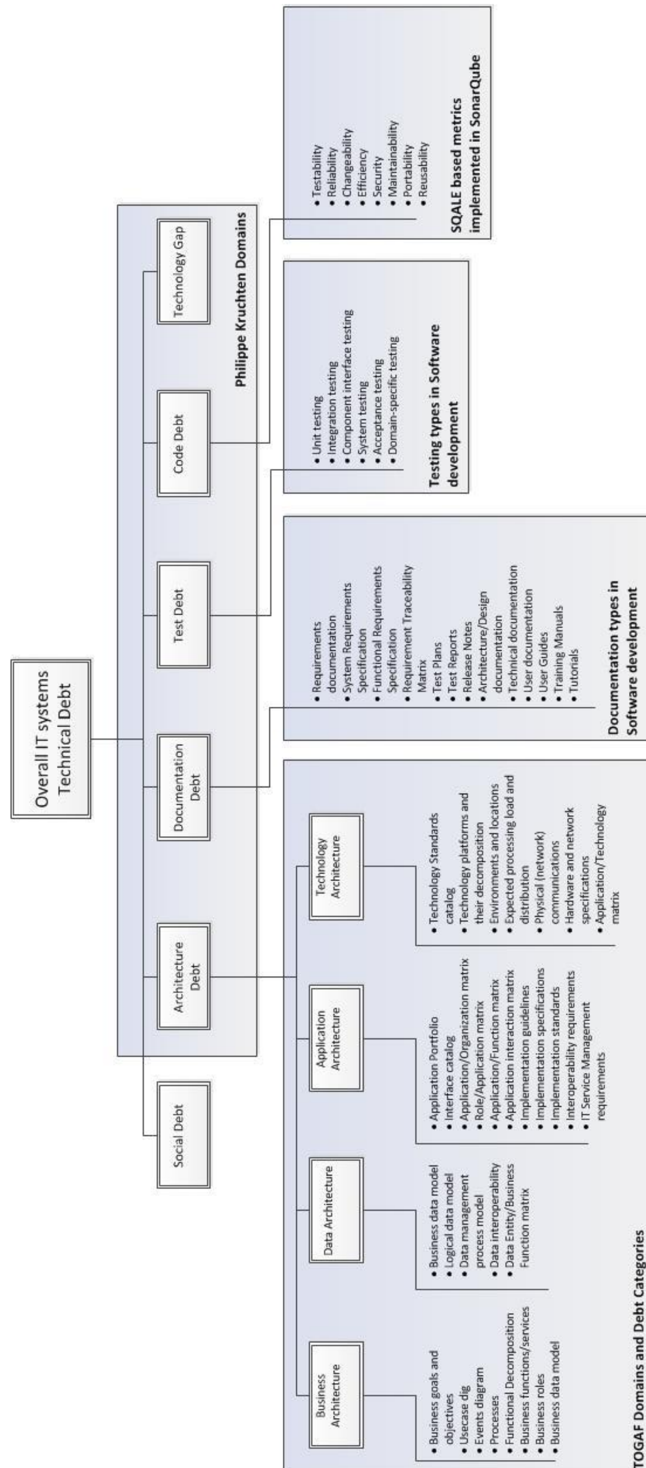
40 Technical Debt Aggregation in Ecosystems
John D. McGregor, J. Yates Monteith, and Jie Zhang; 2012 IEEE

41 Organizing the Technical Debt Landscape - Clemente Izurieta, Antonio Vetrò, Nico Zazworka, Yuanfang Cai, Carolyn Seaman, Forrest Shull; MTD 2012, Zurich, Switzerland

42 The Squale Model – A Practice-based Industrial Quality Model;
Karine Mordal-Manet, Françoise Balmas, Simon Denier, Stéphane Ducasse, Harald Wertz, Jannik Laval

Appendixes

1. Model taxonomy



2. Taxonomy definitions

It's a part of the whole taxonomy definitions documents developed during the theoretical stage of the research.

Domain	Subtype	Definition	Source	Priority	Examples
1. Architectural debt Debt related to high-level technologies, approaches, documents involved in system development. Subcategories are derived from outputs on several stages of TOGAF ADM cycle.					
Business architecture					
	Business goals and objectives	This defines the ways in which a service contributes to the achievement of a business vision or strategy. • Services are associated with the drivers, goals, objectives, and measures that they support, allowing the enterprise to understand which services contribute to similar aspects of business performance. • This also provides qualitative input on what constitutes high performance for a particular service.	1.2	2	
	Use case diagram	Relationships between consumers and providers of business services. • Business services are consumed by actors or other business services and the Business Use-Case diagram provides added richness in describing business capability by illustrating how and when that capability is used. • They help to describe and validate the interaction between actors and their roles to processes and functions. • As the architecture progresses, the use-case can evolve from the business level to include data, application, and technology details. Architectural business use-cases can also be re-used in systems design work.	1.2	1	
	Events diagram	This depicts the relationship between events and process. • Certain events - such as arrival of information (e.g. a customer's sales order) or a point in time (e.g. end of fiscal quarter) cause work and actions to be undertaken within the business. The Event Diagram is an Explorer type diagram that shows Business Events and the Business Processes that they trigger, and also where the Business	1.3, 1.2	2	

	Process flow diagram			1
	Business Interaction Matrix / Functional Decomposition	It shows on a single page the capabilities of an organization that are relevant to the consideration of an architecture. • By examining the capabilities of an organization from a functional perspective, it is possible to quickly develop models of what the organization does without being dragged into extended debate on how the organization does it.	1.2	1
	Business functions /services			2
	Business roles			2
Data architecture				
	Business data model			3
	Business data model			3
	Logical data model			1
	Data management process model			3
	Data interoperability			2
	Data Entity/Business Function matrix	Relationships between systems (i.e., application components) and the data entities that are accessed and updated by them. • Systems will create, read, update, and delete specific data entities that are associated with them. For example, a CRM application will create, read, update, and delete customer entity information.	1.2	1
Application architecture				
	Application Portfolio [Catalog]	Needed to identify and maintain a list of all the applications in the enterprise. This list helps to define the horizontal scope of change initiatives that may impact particular kinds of applications. An agreed Application Portfolio allows a standard set of applications to be defined and governed. It contains the following meta-model entities: •Information System Service •Logical Application Component	1.2	2

		•Physical Application Component		
	Interface catalog	Scope and document the interfaces between applications to enable the overall dependencies between applications to be scoped as early as possible. It contains the following meta-model entities: •Logical Application Component •Physical Application Component •Application communicates with application relationship	1.2	1
	Application/Organization matrix	Relationship between systems (i.e., application components) and organizational units within the enterprise. • The mapping of the Application Component-Organization Unit relationship is an important step as it enables the following to take place: – Assign usage of applications to the organization units that perform business functions – Understand the application support requirements of the business services and processes carried out by an organization unit – Support the gap analysis and determine whether any of the applications are missing and as a result need to be created – Define the application set used by a particular organization unit	1.2	2
	Role/ Application (System) matrix	Relationship between systems (i.e., application components) and the business roles that use them within the enterprise. • The mapping of the Application Component-Role relationship is an important step as it enables the following to take place: – Assign usage of applications to the specific roles in the organization – Understand the application security requirements of the business services and processes supporting the function, and check these are in line with current policy – Support the gap analysis and determine whether any of the applications are missing and as a result need to be created – Define the application set used by a particular business role; essential in any move to role-based computing	1.2	1
	Application/Function matrix	The purpose is to show relationship between data entities and business functions	1.2	2

		within the enterprise.		
	Application interaction matrix			2
	Implementation guidelines			2
	Implementation specifications	See Implementation guidelines.		3
	Implementation standards			1
	Interoperability requirements			2
	IT Service Management requirements			2
Technology architecture				
	Technology Standards catalog	This documents the agreed standards for technology across the enterprise covering technologies, and versions, the technology lifecycles, and the refresh cycles for the technology. It contains the following meta-model entities: •Platform Service, Logical Technology Component, Physical Technology Component	1.2	1
	Technology platforms and their decomposition	Depicts the technology platform that supports the operations of the Information Systems Architecture. • The diagram covers all aspects of the infrastructure platform and provides an overview of the enterprise's technology platform.	1.2	2
	Environment and locations	Depicts which locations host which applications • Identifies what technologies and/or applications are used at which locations • Identifies the locations from which business users typically interact with the applications. • It should also show the existence and location of different deployment environments – including non-production environments, such as development and pre-production.	1.2	1
	Processing Diagram / Expected processing load and distribution	Focuses on deployable units of code/configuration and how these are deployed onto the technology platform. • The Processing diagram addresses the following:	1.2	1

		– Which set of application components need to be grouped to form a deployment unit – How one deployment unit connects/interacts with another (LAN, WAN, and the applicable protocols) – How application configuration and usage patterns generate load or capacity requirements for different technology components • The organization and grouping of deployment units depends on separation concerns of the presentation, business logic, and data store layers and service-level requirements of the components.		
	Physical (network) communications	The purpose of this diagram is to show the "as deployed" logical view of logical application components in a distributed network computing environment. • The diagram is useful for the following reasons: – Enable understanding of which application is deployed where – Establishing authorization, security, and access to these technology components – Understand the Technology Architecture that support the applications during problem resolution and troubleshooting	1.2	2
	Hardware and network specifications			2
	Application[system]/Technology matrix	The System/Technology matrix documents the mapping of business systems to technology platform. • The System/Technology matrix shows: – Logical/Physical Application Components – Services, Logical Technology Components, and Physical Technology Components – Physical Technology Component realizes Physical Application Component relationships	1.2	2

3. Estimation guidelines

#	Domain, debt category	Guidelines for estimation debt principal	Guidelines for estimation of interest
1	BA, Use Case diagram	The estimations are per business function. For removing: (outdated use cases): 1 hour; For adding: 4 hours; Refer the Rest Contract Specifications for high level scenarios. For updating: 2 hours.	Interest is the hours spent in discovering the functionality offered as part of the use case every time this use case has to be reused / changed or new flow has to be added. Interest will be incurred on missing and incorrect scenarios. 10% of (debt missing scenarios + debt incorrect scenarios)
2	BA, Process Flow diagram	This will be applicable for BPM / long running processes. For removing: (outdated processes) 0.5 hours per process For adding: For BPM, probably can be derived from TIBCO designer or BPEL. Can be based on the number of steps involved. 2 hours per 5 steps For Java, this will have to be derived based on design documents / code 4 hours per 5 steps For updating: 2 hours per 5 steps	
3	BA, Business Interaction Matrix	Overview of service provider and consumer. Facilitates service governance. Estimations will be based on number of services. For removing (outdated interactions): 0.5 hour per service For adding / updating: Additions or corrections can be made based on contracts / configurations done (SSL MA), design documents, other knowledge within the teams (Change and Run) 2 hours per service	Interest is the hours spent in discovering the services offered every time a service has to be reused / changed or new flow has to be added. Interest will be incurred on missing and incorrect services. 20% of (missing services + incorrect services)

4	DA, Logical Data Model	There is no enterprise data model within AAB as of today. So, the below estimations are being made per application / business function as the data models are normally created / maintained at this level. For removing(outdated data models): 2 hours per business function For adding: Refer the DDLs (table creation scripts). Some relationships may be within the applications / programs accessing the tables. 10 hours per business function For updating: 3 hours per business function	
5	DA, Data Entity / Business Function matrix	Overview of relationship between Data Entities and Business functions. For removing(outdated relationships): 1 hour for 5 relationships For adding / updating: Refer design documents / application code for deriving this relationship 2 hours for 5 relationships	
6	AA, Interface Catalog	Removal (outdated): Need not be estimated for. Should be much less. Adding / updating existing interface descriptions: Can be derived from design documents. 1.5 hours per business service	0.25 hr. per business service spent on discovering the interface not present in the catalog
7	AA, Implementation Standards	Non-conformance to standards will be the major debt under this category. For adding standards: This type should not be captured as per application. For non-existent standards, adding standards is a debt. This will be based on the platform for which these standards are missing. There should be standards for every single building block in the SOA solution. E.g.: portal, REST services, ESB, Service implementation. In case of non-conformance: This will also be based on the platform / building block involved. Any non-compliance to standards will incur heavy debts, mostly under the reckless type.	

8	AA, Role / System Matrix	Overview of relationship between Roles and Systems. For removing(outdated relationships): 1 hour for 8 systems For adding / updating: Additions or corrections can be made based on design documents, application configurations (role task configurations), other knowledge within the teams (Change and Run) 2 hours per 5 systems
9	TA, Technology Standards Catalog	Like Implementation standards, the major debt under this category will be non-compliance. Adding standards: This type should not be captured as per application. For creating completely new standards. approx. 800hrs. In case of non-conformance: This will depend on the size of the business function (function points for now). 1 hour for every 1 FP with non-conformance
10	TA, Processing Diagram	The estimations will be per environment. E.g.: Internet, GHIA, JAVABATCH, TIBCO Adding diagrams: 24 hours; Removing diagrams (outdated): 2 hours; Updating diagrams: 8 hour.
11	TA, Environment and locations	The estimations will be per environment. E.g.: Internet, GHIA, JAVABATCH, TIBCO Adding diagrams: 2 hours; Removing diagrams (outdated): NA. Should be very less Updating diagrams: 1 hour.

4. Model data collection results

Following fields were eliminated:

1. Debt Type, Debt Status, TD domain
2. System Language, System Business Owner, System IT Owner

#	Artefact Name	Artefact Type	F. area	TD - subdomain	TD - Category	User Added	System Name	Debt, hours	Add. Cst, hour	Occurrence p. year	Prob / hours
1	XXX001	6. BAI	2L	BA	Use Case diagram	WA2126	SYS01	72	6.8	4	27.2
2	XXX001	6. BAI	2L	BA	Business Interaction Matrix	WA2126	SYS01	34	6.8	4	27.2
3	XXX001	6. BAI	2L	AA	Interface Catalog	WA2126	SYS01	25.5	4.25	4	17
4	XXX001	6. BAI	2L	AA	Role / System Matrix	WA2126	SYS01	0.5	1	4	4
5	XXX001	6. BAI	2L	BA	Use Case diagram	JI0108	SYS01	24	2.4	4	9.6
6	XXX001	6. BAI	2L	BA	Business Interaction Matrix	JI0108	SYS01	34	6.8	4	27.2
7	XXX001	6. BAI	2L	AA	Interface Catalog	JI0108	SYS01	25.5	4.25	4	17
8	XXX001	6. BAI	2L	AA	Role / System Matrix	JI0108	SYS01	0.5	1	4	4
9	XXX040	6. BAI	3iD	BA	Use Case diagram	JI0108	SYS01	2	0	1	0
10	XXX040	6. BAI	3iD	BA	Business Interaction Matrix	JI0108	SYS01	2	0.5	1	0.5
11	XXX040	6. BAI	3iD	DA	Logical Data Model	JI0108	SYS01	6	2	1	2
12	XXX040	6. BAI	3iD	AA	Implementation Standards	JI0108	SYS01	16	4	1	4
13	XXX040	6. BAI	3iD	BA	Use Case diagram	WA2126	SYS01	4	0.4	1	0.4
14	XXX040	6. BAI	3iD	BA	Process Flow diagram	WA2126	SYS01	0	0	1	0
15	XXX040	6. BAI	3iD	BA	Business Interaction Matrix	WA2126	SYS01	2	0.4	1	0.4
16	XXX040	6. BAI	3iD	AA	Implementation Standards	WA2126	SYS01	16	4	1	4
17	XXX040	6. BAI	3iD	AA	Role / System Matrix	WA2126	SYS01	0.4	1	1	1
18	XXX001 WS	4. Services		DA	Data Entity / Business Function matrix	JI0108	SYS02	1			
19	XXX001 WS	4. Services		AA	Interface Catalog	JI0108	SYS02	19.5	3.25	1	3.25
20	XXX001 WS	4. Services		AA	Implementation Standards	JI0108	SYS02	16	4	1	4
21	XXX001 WS	4. Services		TA	Environment and locations	JI0108	SYS02	1	1	4	4
22	XXX245	4. Services	1P	DA	Data Entity / Business Function matrix	WA2126	SYS01	0.4	1	2	2
23	XXX245	4. Services	1P	AA	Interface Catalog	WA2126	SYS01	1.5	0.25	2	0.5

24	XXX245	4. Services	1P	BA	Business Interaction Matrix	WA2126	SYS01	2	0.4	2	0.8
25	XXX245	4. Services	1P	AA	Implementation Standards	WA2126	SYS03	8	4	2	8
26	XXX509	6. BAI		BA	Use Case diagram	PA2778	SYS03	30	3	2	6
27	XXX509	6. BAI		BA	Process Flow diagram	PA2778	SYS03	12	6	2	12
28	XXX509	6. BAI		BA	Business Interaction Matrix	PA2778	SYS03	4	0.4	2	0.8
29	XXX509	6. BAI		DA	Logical Data Model	PA2778	SYS03	0	0	2	0
30	XXX509	6. BAI		AA	Interface Catalog	PA2778	SYS03	22.5	3.75	2	7.5
31	XXX509	6. BAI		AA	Role / System Matrix	PA2778	SYS03	0.4	1	2	2
32	XXX509	6. BAI		AA	Implementation Standards	PA2778	SYS03	120	40	2	80
33	JAVA BATCH	10. Solution Building Block	4B	AA	Implementation Standards	WA2126	SYS01	40	6	3	18

5. Model feedback questionnaire

Questions	Reply option
1. How long have you been working in ICT sector?	Input number
2. What is your main activities in working time: development, architecture, testing, support, technical design, management, other?	Choose up to 2 main activities
3. How long have you been working at ABN AMRO projects?	Input number
4. If the research question about is technical debt measurement, to what degree do you think using the proposed model will enhance your job performance? 5. To what degree do you think using this model for estimating TD would require much effort to use? 6. To what degree do you think this model is consistent with the existing values, needs and experience related to TD awareness ? 7. If the research question is about technical debt measurement, to what degree do you think you would use this model in future? 8. If the research question is about technical debt measurement, to what degree do you think other colleagues of yours would use this model? 9. To what degree the model is able to measure the amount of TD incurred by system?	Choose one of the options: • Very low degree • Low degree • Average • High degree • Very High degree
10. Do you think there are aspects that are missing in this model?	Input text

11. Did you have any difficulties while understanding the model or filling in the measurement table?	Input text
12. Please estimate how much time it took you to fill in the measurement table (in total or per component)	Input number
13. Please give your comments on following results of comparing data collection: • There are 50% changes in list of TD categories describing XXX040 component. • We found that estimation made by two different people about two different components may be different by 2-3 times.	Choose one of the options: • Model is ambiguous? • Estimation guidelines are not specific enough? • Employees have personal view on TD amount estimation? • Employees have personal different professional experience? • Other

6. Feedback questionnaire results

Question number	WA2126	JI0108	PA2778
1	13	7	6
2	architecture, technical design	development, technical design	development, testing
3	8	7	3
4	High degree	High degree	High degree
5	Low degree	Average	Average
6	High degree	High degree	High degree
7	High degree	High degree	High degree
8	High degree	High degree	High degree
9	High degree	Average	Average
10	There should be some way of capturing the relationship between different categories. For example an incorrect Use case diagram (BA) can influence technical debt in testing.	We only calculated for architectural debt. It will be more clear if others are also done	
11	Not much. I had to refer to the category definitions in repository.	I was able to understand it with detail text	Not really
12	0.5	1	2
13	Employees have personal different professional experience	Employees have personal view on TD amount estimation	--