Lappeenranta University of Technology
School of Industrial Engineering and Management
Degree Program in Computer Science

Dmitrii Poliakov

A SYSTEMATIC MAPPING STUDY ON TECHNICAL DEBT DEFINITION

Examiners : Professor Kari Smolander
M. Sc. Jesse Yli-Huumo

ABSTRACT

Lappeenranta University of Technology
School of Industrial Engineering and Management
Degree Program in Computer Science

Dmitrii Poliakov

A systematic mapping study on technical debt definition

Master's Thesis

79 pages, 11 figures, 3 tables, 2 appendices

Examiners: Professor Kari Smolander
M. Sc. Jesse Yli-Huumo

Keywords: technical debt, mapping study

The goal of this study was to explore and understand the definition of technical debt. Technical debt refers to situation in a software development, where shortcuts or workarounds are taken in technical decision. However, the original definition has been applied to other parts of software development and it is currently difficult to define technical debt. We used mapping study process as a research methodology to collect literature related to the research topic. We collected 159 papers that referred to original definition of technical debt, which were retrieved from scientific literature databases to conduct the search process. We retrieved 107 definitions that were split into keywords. The keyword map is one of the main results of this work. Apart from that, resulting synonyms and different types of technical debt were analyzed and added to the map as branches. Overall, 33 keywords or phrases, 6 synonyms and 17 types of technical debt were distinguished.

Contents

LIST OF SYMBOLS AND ABBREVIATIONS

TD	Technical debt
SLR	Systematic literature review
TDM	Technical debt management
RQ	Research question
TDD	Test driven development

1. INTRODUCTION

In the recent years, there have been many papers describing the problem of neglecting basic software engineering principles due to shortages in project budget [40], urgent needs of the market and therefore lack of the time [67], or even a plain human factor such as inadvertence [23]. It is commonly suggested that a lot of software projects end or are shut down without any successful results [15]. When investigating causes of the failures, they are often found among the technical problems, such as the increased complexity of the code base [41]. Thus, disregard of the basic principles and non-use of best practices, which led to a poor state of affairs, can be often fatal to a project. This makes the described problem important for most of the practitioners, who are interested in success of their projects indeed. In order to describe this problem a lot of different definitions have been used. One of the most often mentioned is technical debt (TD), which was proposed by Cunnigham via metaphor "*Shipping first time code is like going into debt*" [8]. The metaphor has been proving itself useful during the recent years. It is able to describe the nature of a technical problem succinctly and accurately and makes possible to start searching the appropriate solutions.

However, this metaphor can be used wrong without proper understanding. For instance, broadening the metaphor indefinitely kills its original purpose. A large number of studies have been using this metaphor, but very few of them had the aim to state its definition or shape the metaphor of TD more precisely. It is would be also useful to make explicit the consequences and problems that practitioners are likely to face dealing with TD and identify main characteristics of it. Such information at practitioners' fingertips may allow them to benefit from TD instead of suffering losses.

Therefore, the goal study aims to analyze different approaches to definition of TD metaphor and compile them into several artifacts that are perspicuous for any reader who is interested in this topic. We also try making an attempt to generate a more precise definition of TD, based on our findings.

For this purpose we conducted a mapping study that analyzed all the articles available through the most popular scientific databases. The searched papers contained a reference to the original article of Cunnigham "The wycash portfolio management system" [8] in order to determine how their authors understand and use the proposed TD metaphor in their research.

The remainder of this paper is divided into six sections including this introduction. After this introduction, the second chapter is the description of TD metaphor in form of literature review focusing on researches that made attempts to describe it. The third chapter describes the methodology of used in this research process and our adjustments to it that were made in order to meet the needs of this research. The fourth chapter describes the results. Then, the fifth chapter discusses the results of the research and answers on research questions focusing on keywords that describe TD metaphor. At the end, conclusion chapter summarizes all results and makes suggestions for the future researches on the proposed topic.

2. BACKGROUND

Roots of the technical debt metaphor

The original article was written in 1992 by Cunnigham [8]. According to this article, the developers meet different challenges during the system implementation and the main purpose of this paper was to address these issues. Cunningham described these problems with such words: "*it was not uncommon for a new feature to fit poorly into an existing object architecture … we found that some key implementation ideas were slow to emerge*" [8]. The solution to these problems was described as: "*The ultimate removal of the immature architecture would leave us with a program that had been simplified in the course of adding features-a truly enviable situation. We believe this process leads to the most appropriate product in the shortest possible time*" [8]. In order to give the reader better understanding of the problem and describe it metaphorically Cunningham writes about TD: "*Although immature code may work fine and be completely acceptable to the customer, excess quantities will make a program unmasterable, leading to extreme specialization of programmers and finally an inflexible product. 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. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.*" [8]

Cunningham describes the waterfall model as an attempt to pay the whole possible debt up-front and in-full by spending a lot of time in earlier phases when the goal is to develop the overall architecture of the solution beforehand. This model can be described in financial terms like we pay for everything to preserve it instead of paying for fixing what it is already broken. In this model we have to envisage every possible failure in every vulnerable spot in advance, which is very difficult or impossible in constantly changing world when the requirements for the system can be added, removed or altered completely [8].

When using the original definition by Cunnigham [8], TD can be seen as the cost that needs to be paid for fixing some suboptimal decisions that were made during the system development. This cost can be incurred in additional expenditure when project managers have to hire more programmers to make changes in the project source code or in terms of time when programmers have to spent additional time which is needed to understand the increased complexity of the poor-structured system or effort that has to be applied to continue the project and finish it with success.

McConnell [48] describes TD as: "*a design or construction approach that's expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time).*" A map of the "technical debt landscape" was drawn during the third workshop on managing TD in 2012 [48]. This landscape covers intentional TD (or architectural debt), TD due to a change in context (technological gap), and at the right end TD of a smaller granularity, mostly low internal code quality. On the left of the Figure 1, TD affects mostly the evolvability of the software system, while on the right it affects mainly its maintainability [37].

Fowler [21] describes TD as "*piece of functionality that you need to add to your system. You see two ways to do it, one is quick to do but is messy - you are sure that it will make further changes harder in the future. The other results in a cleaner design, but will take longer to put in place.*" He categorizes TD into quadrant with two dimensions. First dimension separates issues arising inadvertently from decisions that were made strategically. Second dimension separates debt into reckless and prudent categories. Reckless and deliberate debt is the one that is discussed the most in this study. It takes place when practitioners are aware of consequences but they simply cannot afford writing clean code because the lack of time or other reasons, which will be discussed in more details later. Prudent and deliberate debt is common for teams that know how to use TD in their advantage and do it. Oppositely, reckless and inadvertent debt is something that is named "messy code" and may be produced by people who are ignorant of good design practices. Fowler claims that prudent and inadvertent debt is also possible. It may take place when a team of excellent professionals is learning during the system design process. For the time being they think they have done a good job and customer is happy with the

system quality, but after a while they realize what they should have done something different. These four categories can be seen in Figure 2. According to the author's opinion, this quadrant should be used to understand the current situation in your project and correct it based on the quadrant section which we are currently in [21].

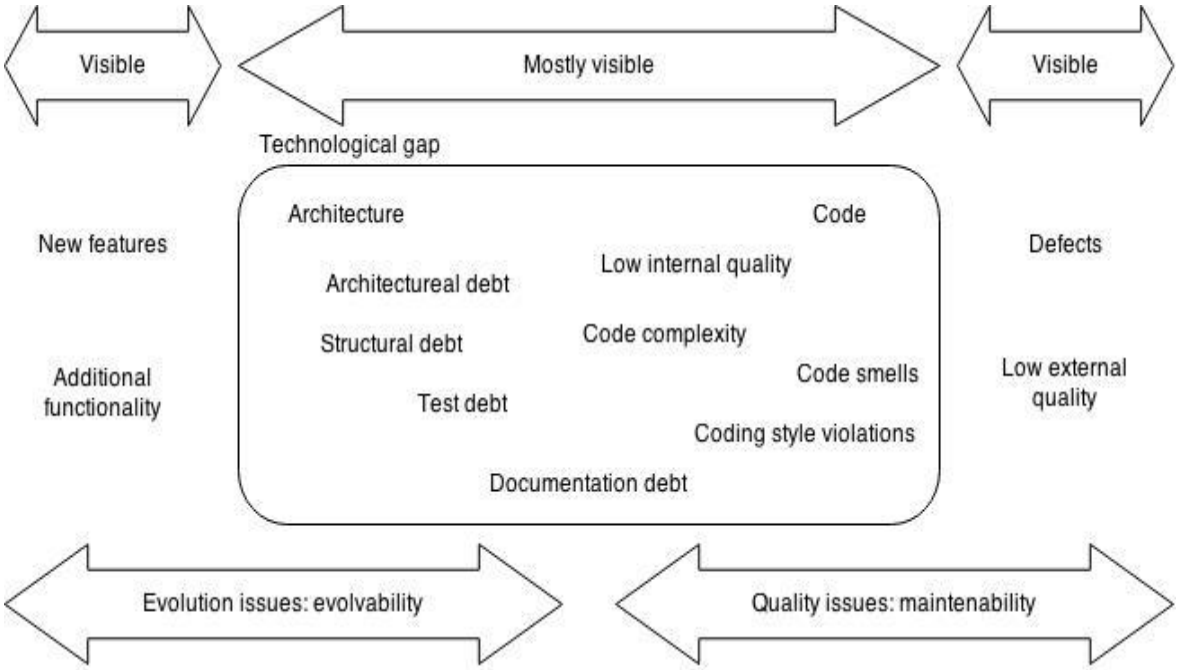

Fig. 1. TD map [37].

It is worth noting that in addition to TD metaphor, there are also alternative descriptions of the existing problem, for example "code smells". "Code smells" (introduced by Fowler and Beck [11]) is an established concept to classify shortcomings in software that follows object-oriented design. Each "smell" is an indicator that points to the violation of object-oriented design principles such as data abstraction, encapsulation, modularity, and hierarchy.

All these descriptions have several commonalities. One of which is that the more TD takes place the worse will be the consequences. In TD terminology it would sound like: the more you go into this debt the more interest you have to pay. For example, to change something in the code that is written without applied encapsulation principle or is not modular the programmer has to pay all the interest first by checking and debugging program in different places instead of introducing new functionality. There can be even a dead end

when a fix in one place leads to two or more broken parts in other places, which has an avalanche-like effect eventually leaving you with a completely unsupportable system. Issues of this size might force you to make the final decision to rewrite the whole system from scratch or close the failed project [1].

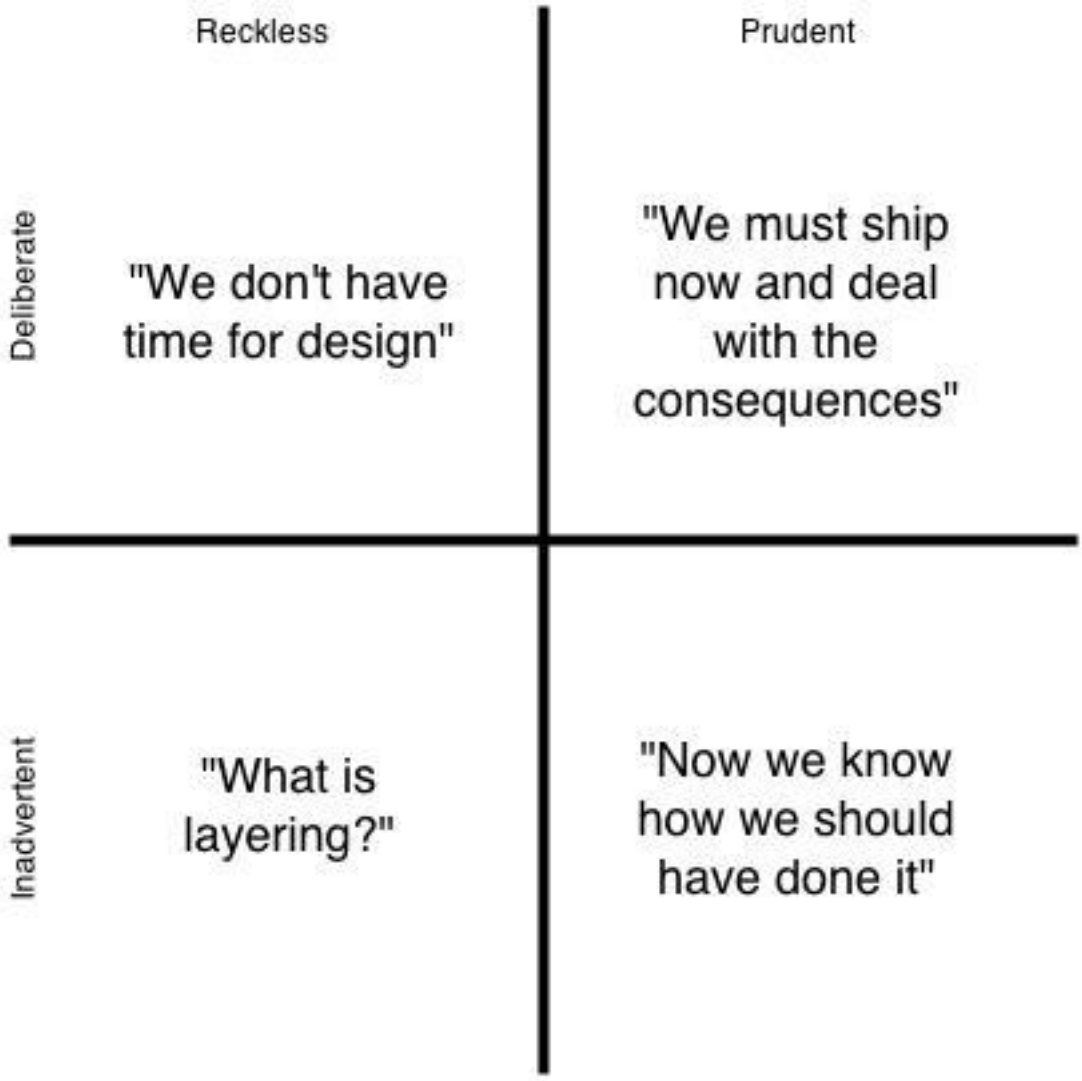

Fig. 2. TD categorization [21].

The success of TD metaphor probably lies in its financial origin. To give people better understanding of the complicated technical problem it is highly desirable to use words that they know and commonly use in their work and life. Basic financial vocabulary like debt and interest payments is well-known by most of people so all this words are appropriate

and understandable when a short description of emerged problem is needed. According to Allman [1], financial debt has three important properties: there is a need to pay eventually back, you have to pay with interests, and there will be consequences if you do not pay your debt in time. TD is in a way similar. Generally, it is necessary to pay back by refactoring or completely rewriting some parts of software system. Until then, people that work with it have to pay interests by being delayed with mysterious problems or urgent necessity to fix something instead of implementing new functionality. Sometimes it even happens that a user has to pay interests for TD taken by developers [1]. Although, unlike financial debt, TD does not have to be paid off entirely. There are very few systems that do not need any fixing or refactoring [1].

The causes and effects of technical debt to software

According to Edith et al [65], companies have several reasons to go into TD. These reasons can be compared with TD quadrant suggested by Fowler [21], which is depicted in Figure 2.

- Pragmatism and prioritization – when you have to make sub-optimal decisions because of situation on the market. In this case, the implementation of critical functions can be considered more important than overall quality. This can be Deliberated reckless or deliberated prudent TD depending on context and further decisions. For example, will it be paid eventually in the future or not?
- Because of deficient process that is employed by the development team – in general, existence of review code practices, meetings (as it is in agile development methodology) or other collaborative techniques can significantly decrease amount of TD. Otherwise when you make your own individual decision it is easier to take shortcuts and defer tasks that increase TD. Thus, individual decisions may cause reckless and inadvertent TD.
- Certain attitude of developers can also cause TD. Elm [16] describes one of the common reasons for unmanaged debt to be "*programmer and management hesitance to improving code, for fear of introducing new problems: 'If it ain't broke, don't fix it*". TD caused by such kind of attitude without any external necessity like market needs or customer demands is reckless and deliberate debt.

- Ignorance – “the inability of the developers to develop high quality applications” (Cast [6]). This is similar to Reckless and inadvertent segment in Figure 2.

As for affecting software, resulting map from TD workshop 2012 [38], [39] depicted in Figure 1 divides this into two global issues: evolvability and modifiability. According to this map, the existence of TD in a project can be determined by presence or absence of the particular symptoms. These symptoms are depicted on Figure 1 in the area of technical gap, and right and left of it in the visible areas. Each of these symptoms may be visible or invisible and at the same time refer more to the problem of software evolution or its maintenance. Traditionally, characteristics such as the addition of new functionality are more relevant to the problem of evolution whereas such parameters as internal and external quality, code complexity and code style violations affect on the ability to effectively maintain software. Thus, as it is mentioned above, the allowance of TD can facilitate software development process for a while but decreasing its maintainability in the long term at the same time.

Technical debt and software lifecycle

According to IEEE Standard for Developing Software Life Cycle Processes [31], there are five general groups of related activities.

Every software lifecycle starts from project initiation, which is a part of project management group. Project monitoring and control, and software quality management are two other parts of project management group. These two are necessary for each project iteration.

The second group is pre-development activities. This group consists of the processes that must be performed before software development can begin. Concept exploration can be a good example of such kind of activities.

The third group is the development itself. It includes the activities performed during the development and enhancement of a software project.

The activity groups of the Post-Development Section include activities performed during the development and enhancement of a software project. The group of retirement activities here ends the cycle of iterations for software in development.

The integral activities group is in the middle of figure 3. This group is the support section, which includes activities that are needed to successfully complete project activities. These activities are utilized to assure the completion and quality of project functions.

TD can also be classified from the perspective of the software lifecycle, which means that several types of TD are related to lifecycle phases. The same effect of taking shortcuts can happen in several stages of software development lifecycle. For example, design debt, testing debt, defect debt, documentation debt [56] and possibly other debts are mentioned to be part of the software lifecycle. This can be seen as one of the reasons why the current definition of TD has multiple aspects.

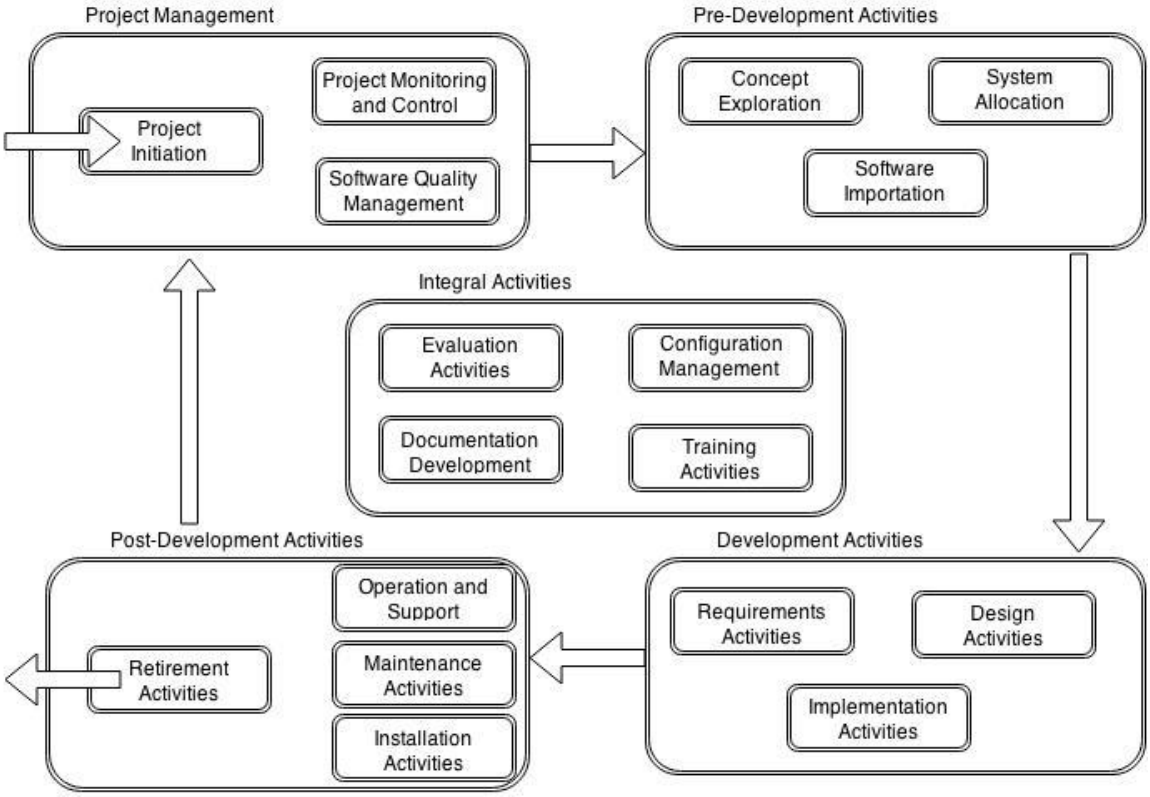

Fig. 3. Classification scheme iterative building process [31].

3. MAPPING STUDY

Mapping study origins

Secondary studies are drawing more and more attention of scientific community because, when the number of reports in a field is constantly growing, it becomes important to provide an overview of the existing scientific sources to structure them and prepare the solid basis for further studies of the subject [14], [28]. Evidence based medicine is a good example of utilizing secondary studies, especially systematic literature review (SLR) approach [34]. SLR approach consists of going in dept of existing primary studies in order to review their essence and describe the used methodology and received results [34]. According to Kitchenham et al. [34], this approach is also common for such fields like criminology, social policy, economics, nursing etc. where just an aggregation of all existing evidences about some predefined subject is not enough because there is a need of rigorous method which helps to avoid any bias and inaccuracy in source selection and review. Keele [33] represents this approach as "*a means of evaluating and interpreting all available research relevant to a particular research question, topic area, or phenomenon of interest*".

However, SLR is usually used for reviewing quantitative and empirical researches whereas in software engineering the number of qualitative researches is currently growing and it is necessary to find more suitable method for summarizing emerging scientific reports [13]. One of such methods is a systematic mapping study, which "*provides an overview of a research area, and identifies the quantity and type of research and results available within it*" [54].

There are differences between a mapping study and SLR. Keele [33] summarizes them as following:

- Research questions of mapping studies are usually wider and often numerous.
- The search results for mapping studies are likely to return a big number of studies. However it is not so problematic as it is for SLR because the aim is the broad coverage rather than narrow focus.

- The data extraction process in case of mapping studies is also different from the process existing for SLR and is rather classification and categorization than extraction. The main goal of this step is to categorize papers in order to answer all the research questions as it is in case of SLR but here the focus is on checking numerous studies without spending too much time on each particular one.
- The purpose of the analysis stage is summarizing the gathered data to answer the research questions. Since the number of selected studies is large, it is unlikely to include in depth analysis techniques such as meta-analysis and narrative synthesis. Totals and summaries as well as graphical representations of study distributions by classification type are the main techniques for mapping studies.

Figure 4 shows the process of this research. According to Petersen et al. [54], the rigorously prepared mapping study has to have predefined research questions before continuing further. The next step is to find all available scientific sources that comply with the predefined goal. One of the main differences of mapping study from SLR is that there is no need to consider specific criteria which the found scientific sources have to comply with. This way the area of research can be artificially narrowed and the main idea of mapping study is to provide a broad overview. The application of selection criteria happens only after gathering a broad selection of scientific sources and this process follows the clear predefined list of criteria to avoid any possible bias. After defining the core selection all sources need to be classified and unlike the previous steps the classification schema here is usually defined iteratively, straight during the process of keywording. In other words, if there have been found some words which do not fit to any currently existing categories a new one can be defined to cope with this situation. The Figure 4 below illustrates this process.

The enclosing step in this process is the visualization of results according to defined classification schema. This visualization usually takes form of bubble map or any other which is appropriate for this task. After that, the received results are to be discussed and the research questions are to be answered.

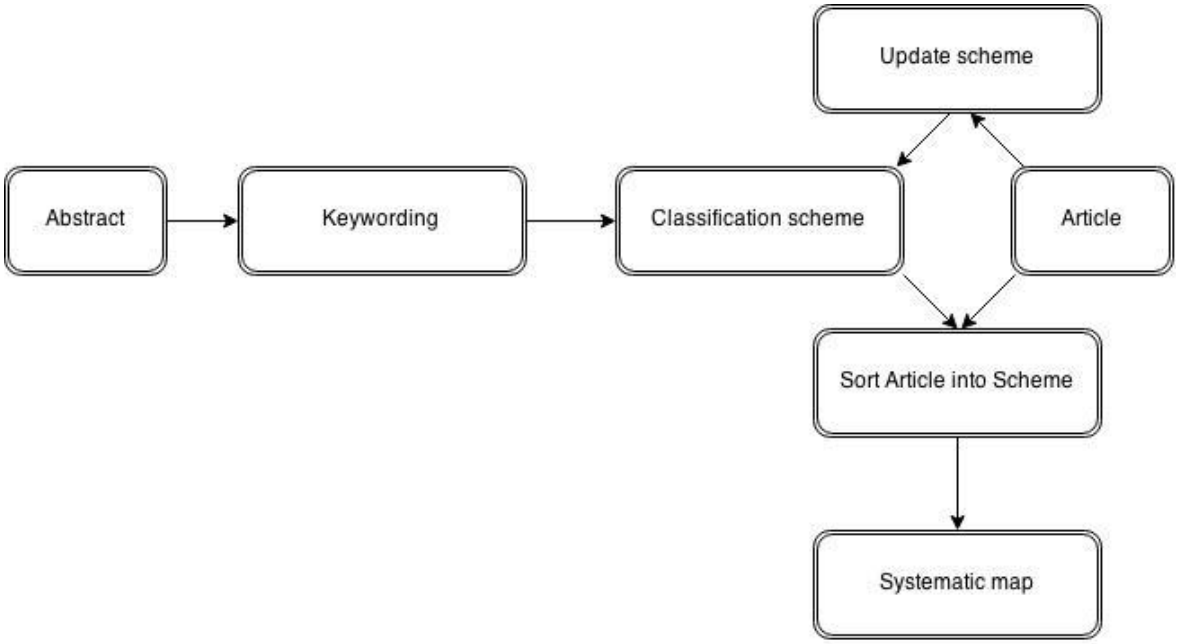

Fig. 4. Classification scheme iterative building process [54].

Mapping study process description

The procedure of this study followed the basic guidelines for mapping study methodology provided by different authors, in particular Kitchenham and Charters [4]. Several steps have been performed to achieve research goals and answer on defined question. The steps of this research are shown on the figure 5 below.

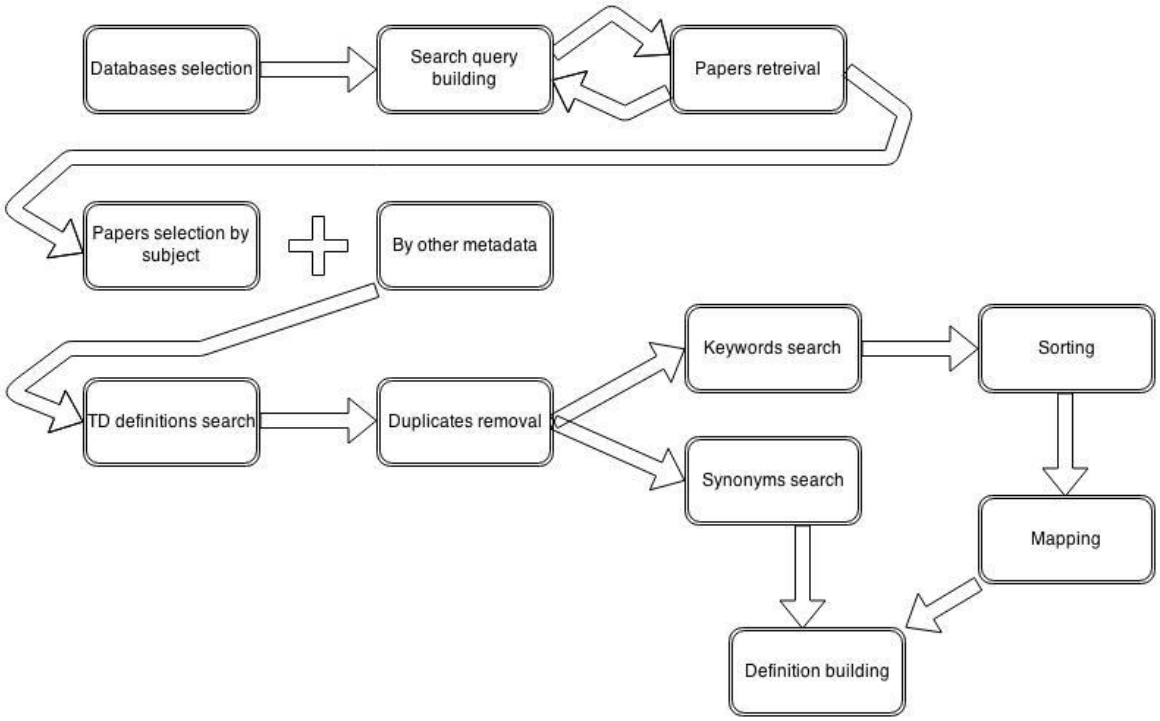

Fig. 5. Study process diagram.

Before going into mapping study process we identified four research questions. These questions become the goal of the current research.

- RQ1: What technical debt is?

This is the central question of this study. Understanding of the TD essence provides the basis for answering the following research questions. From practitioners' point of view, answer on this question allows more accurate assessment of the situation in ongoing project, correction of the employed strategy and process and eventual success in the long term perspective with use of the TD as a tool to accelerate development in any crucial moment.

- RQ2: Which synonyms of the technical debt metaphor are there?

TD, apparently, is not the only one metaphor that describes the matter under consideration. Synonymous definitions availability can help shaping the TD metaphor itself as well as define its frames by noticing differences between TD and other definitions.

- RQ3: What types of technical debt are mentioned in scientific works?

Differentiation of TD types can help to ascertain the cause of problems encountered in software project, determine the stage of the project life cycle during which the debt was taken and thus establish his whereabouts and to understand reasons of its occurrence.

- RQ4: Is it possible to build the general definition of technical debt metaphor?

The possibility of constructing general definition of TD would be crucial for communication between practitioners and researchers because it would allow defining TD similarly and thus enforcing the use of Cunnigham's metaphor more precisely avoiding possible misunderstandings.

The appropriate databases of scientific publications were selected with regard to their scientific community acceptance and rating. Six scientific bases were included in the final list:

- IEEE Xplore (http://ieeexplore.ieee.org/Xplore/dynhome.jsp).
- ACM Digital Library (http://portal.acm.org/dl.cfm).
- Springer Link (http://link.springer.com/).
- Science Direct. (http://www.sciencedirect.com).
- Ebsco (http://search.ebscohost.com).

As an addition to these databases the decision was made to perform further search iteration in Google Scholar service to ensure that no relevant scientific source is missed.

The query for database search incorporates words from Cunnigham's publication name and logical operator AND, which ensures that only papers with possession of exact phrase will be selected. Several queries before the final one have proven that the name of Cunnigham's article is unique among existing articles in selected databases, so there was no risk to stuck on irrelevant results during the research process.

All available papers were added in the Refworks reference management tool, which was supposed to store them in well-organized manner and was able to transform the list of sorted references in any bibliography according to specified criteria.

As it is shown in the Figure 5, there were several rounds of papers selection. The first round represents retrieval of raw materials without application of any selection criteria. This round was finished by retrieving of one hundred fifty five papers from five databases and additional search performed with Google scholar search engine. On the second round all papers were put into one folder and during this process all duplicates between databases were deleted. This process was performed in the order which is shown in the Figure 5 from IEEE Xplore database to the results from Google scholar. After that, the selection criteria were applied to the rest of papers. Metadata and introduction of each paper were analyzed in order to ensure that all criteria are suitable for each piece of work. The used criteria are listed below:

- The language in which each work is written must be English.
- The full text of work and its metadata must be freely accessible.
- The main subject of this work is TD or TD takes considerable part of work's focus.

All the results of each selection phase are presented on Figure 6.

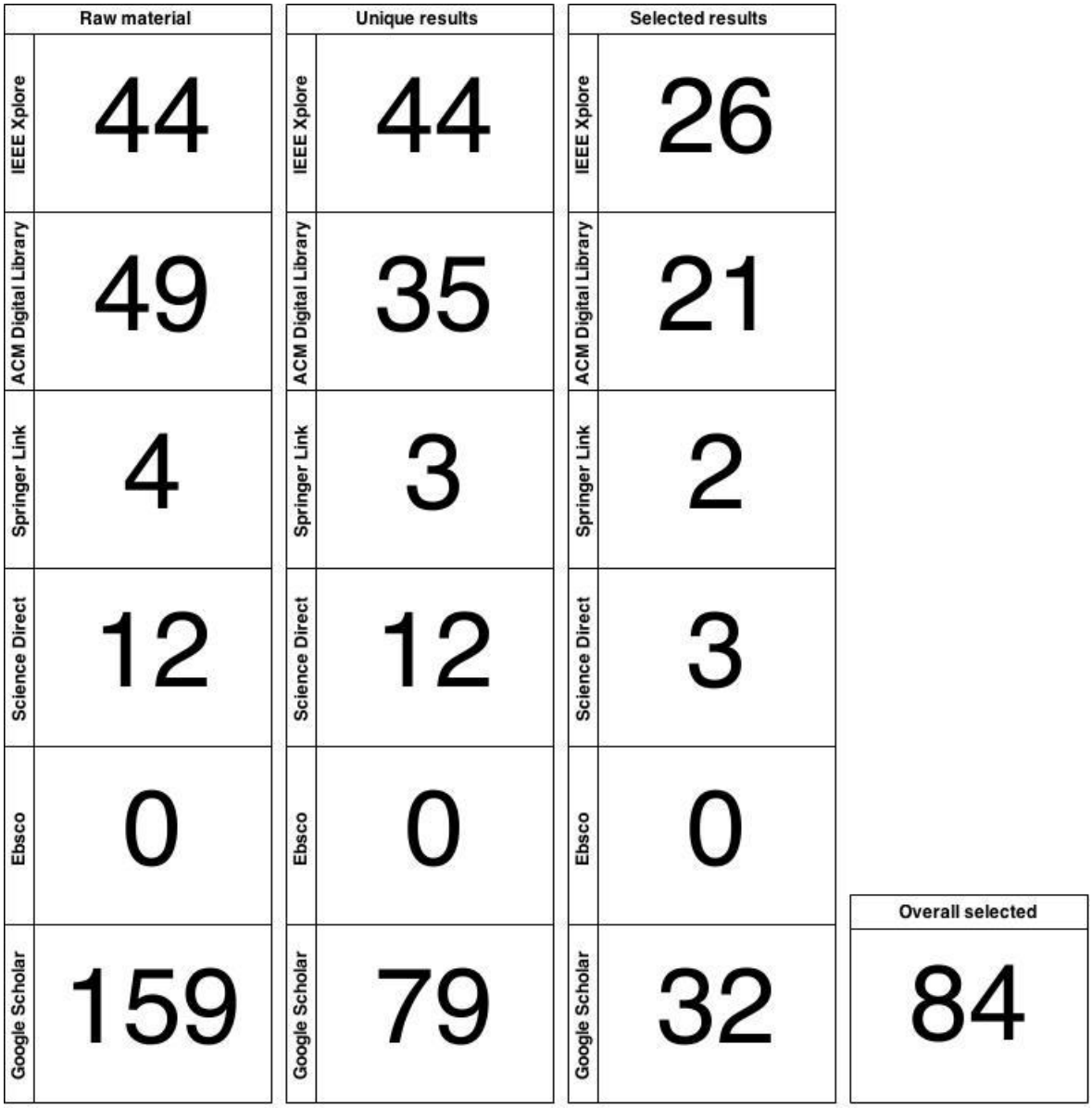

Fig. 6. Selection results.

All the papers were ordered by their date of publication, source and the object of study in order to analyze them easier. The analysis of works consisted of skimming the selected articles to find all used definitions of TD and determine what synonyms are used in scientific community to name this metaphor differently. Also, mentioned types of TD were extracted and sorted. Every piece of handwriting which was interesting in scope of this research was copied in a separate table. All duplicates were removed to ensure validity of the next step.

The process of keywords and extraction was performed iteratively. First iteration was performed manually. The following steps were taken for each source:

- Metadata evaluation. This step will help filtering out all the sources for which TD is not the main focus, saving this way time by avoiding analysis of useless data sources in depth.
- Extracting main idea of study from abstract. Evaluation of abstract provided an opportunity to assess the necessary depth of skimming each paper by considering of performed study process and its results.
- Introduction and first chapter skimming (usually the part which introduced definitions). Manual search of TD definitions by skimming the text and identifying every interesting sentence or phrase. Each phrase without any further analysis was noted in a separate table in order to undergone analysis later.
- Other part skimming (only for those studies which had definition of TD as a research question).

Second iteration was the raw analysis of retrieved material:

- Cycled process of creating classification schema by evaluating every retrieved piece of information and sorting each piece among created nodes of this schema was performed.
- The phrases that seemed interesting at first glance but were not suitable for purpose of the study were sorted out on this step.
- On this step phrases were divided into defining the TD, synonymous with TD and determining the type of TD. It was done in order to answer the basic questions of this study.
    - Phrases that intended to determine TD, succinctly describe it as a separate definition, were sorted to the group of the TD definitions. Each definition from the resulted table was split on separate keywords and all the keywords were sorted by the frequency of occurrence. Because keywords was too numerous to show them in one list, it was decided to separate them into three groups. The main group consists of keywords that occurred five or more times, the secondary group consists of keywords that occurred less

than five times and the tertiary group consist of keywords that occurred only once.

- Separately introduced terms that from the first sight described the similar to TD concept were sorted to the synonyms group.
- All definitions found in the selected sources that possessed a certain quality which indentified their uniqueness from others but still admitted that they are TD were attributed to types or kinds of TD.

The third iteration was performed automatically with use of full-text search tools.

- To determine TD multiple additional searches have been made. Their results are not presented as a separate part of this work because they did not provide any useful information for research questions directly. However, they aided greatly in the construction of classification scheme and subsequent decomposition of certain TD definition on keywords and their following classification.
- An additional search of the types of TD was made through the whole set of selected papers in order to determine frequency of occurrence.

Visualization of the results was the last part of the analysis process. In order to visualize the final map was drawn. This map is described in the next chapter.

4. RESULTS AND ANALYSIS

This mapping study was performed according to the procedure described in the chapter three. In this chapter, results of retrieved data analysis are described. To begin with, the classifications of selected studies by source, type and publication year are provided. The rest of this chapter provides results of TD definition's separation into keywords, synonyms and TD types extraction. At the end of this chapter all synonyms, TD types and resulting keywords are visualized in map.

Classification by source

The pie chart below represents the distribution of articles retrieved from the databases. It is worth saying that most of the papers that were found with the use of Google Scholar search engine were overlapping with the papers from the other databases, which makes the picture look different from the one depicted below if all duplicates are removed.

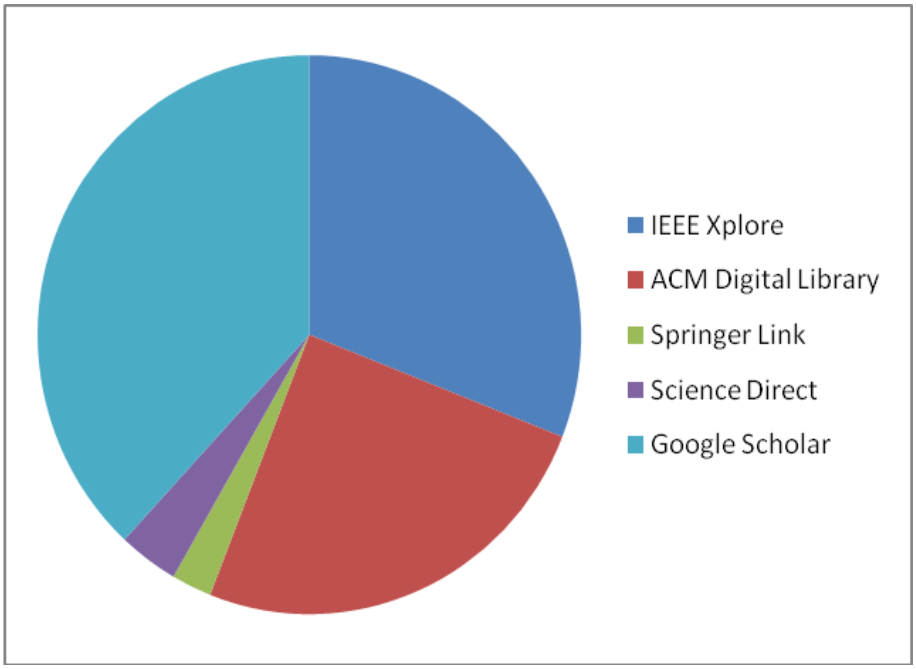

Fig. 7. The shares of scientific works from different databases.

Classification by type

Selected papers were divided into 4 types: articles, workshop reports, conference reports and symposium reports. It can be seen from the pie-chart that articles group has the largest share. Overall, 32 articles were selected for this research. The second largest group is workshop reports. It incorporates 28 pieces of writing. Conferences and symposiums groups have 19 and 4 works accordingly.

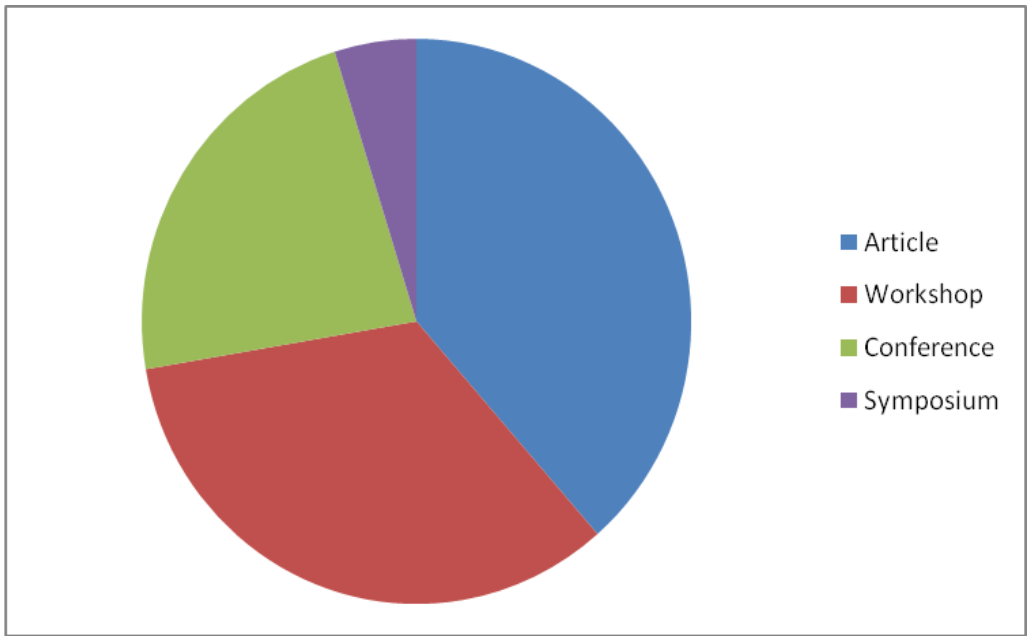

Fig. 8. Types of scientific works.

Classification by publication year

The curve below represents amount of works among the selected ones and their distribution by publication years. Despite the fact that there were no restrictions in the search process concerning publication year, except one natural restriction that citing paper cannot be written earlier than Cunnigham's work, almost all papers that cite "WyCash Portfolio Management System" were written in between 2009 and 2014 (except one paper in 2006). The mapping study of Zengyang et al. shows similar curve in distribution of publications. According to Zengyang et al. [42], the reason for this could be that the MTD workshop was initiated in 2010 and this workshop raised the attention on TD and the awareness of managing TD.

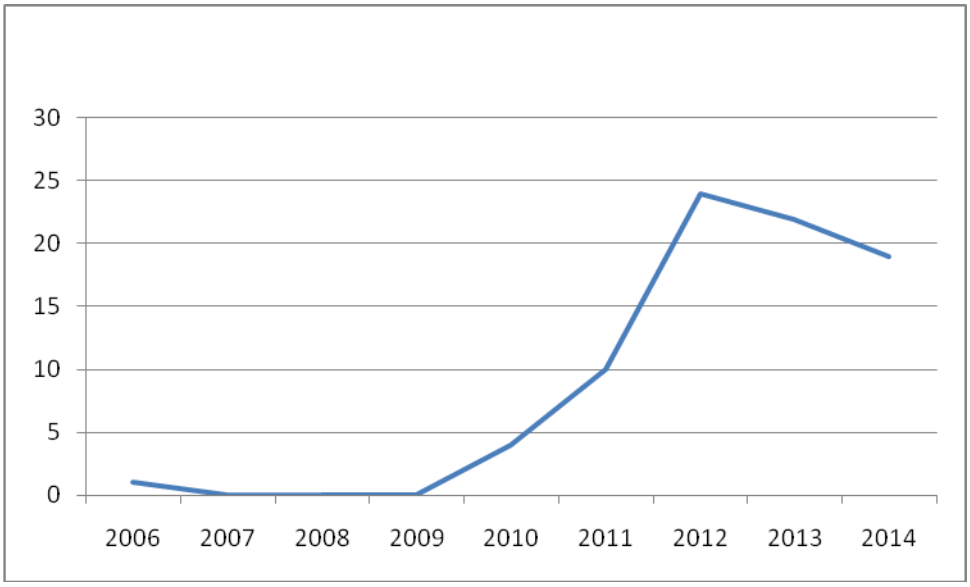

Fig. 9. Year of publication distribution curve.

As the Figure 9 shows, the perceived value of metaphor for practitioners and researchers is growing. More and more of them consider it necessary to study the problems captured by the TD metaphor.

Extracted definitions

Appendix 2 presents all definitions and synonyms in form of table that were retrieved from selected papers in raw, unchanged form. This table presents only unique definitions that have been found during the search process. Based on research conditions all 107 papers had definition of TD cited or rephrased from Cunnigham's article in addition to others or as the only one definition used in the whole paper, therefore their duplications were excluded from the table. Each definition listed in Appendix 2 has a link to source paper the list of which is presented in Appendix 1.

Several works present multiple definitions of TD. Basically, these are works with the aim to determine what TD is. These studies thereby are the most valuable for the current study, because it aims to combine such efforts within the secondary mapping research. The biggest number of TD or its synonym's definitions was extracted from sources [26], [38],

[27], [43], [60], and [63]. For example, Lim [43] describes the interview of practitioners about TD the purpose of which was to define TD.

Partition of definitions into keywords

When we had retrieved all the definitions from the selected papers, all of them were partitioned into keywords according to the procedure which is described in the methodology section. The occurrence of each keyword is observable from Table 1. Overall, thirty three keywords or phrases were extracted from TD definitions.

Table 1. Keywords that were retrieved from TD definitions.

Keyword	Source
cost / budget	S68, S69, S11, S1, S1, S8, S59, S59, S78, S63, S8, S54, S49, S24, S24, S27, S22, S50, S39, S39
short/long term	S11, S11, S6, S12, S1, S8, S85, S78, S43, S40, S43, S22, S86, S53, S50, S35, S41, S41
tradeoff/compromise	S68, S11, S11, S4, S14, S19, S86, S78, S8, S61, S43, S51, S37, S50, S35, S35, S41
Quality	S68, S2, S4, S20, S15, S15, S15, S59, S59, S8, S61, S38, S39, S35, S26, S36, S36
Maintenance	S10, S7, S19, S8, S78, S63, S43, S33, S45, S27, S27, S22, S48, S23, S50, S50
Shortcut	S11, S8, S43, S1, S49, S40, S33, S43, S22, S23, S50, S35, S35, S41
time / deadline / release date / reach of markets	S68, S2, S1, S1, S8, S61, S51, S53, S37, S50, S50, S35, S35
best practices	S1, S15, S58, S70, S33, S45, S27

Keyword	Source
refactoring/rework	S6, S17, S33, S33, S85, S53
"right" / "wrong" or "dirty" code	S69, S12, S13, S46, S27, S53, S34
postponing changes	S22, S50, S35, S34, S46
Architecture	S58, S45, S51, S50, S27
business benefits	S35, S43, S35, S35
Gap	S20, S45, S36
Complexity	S20, S24, S32
Dimensions	S2, S14
Value	S54, S56
modularity violation	S2, S62
internal consistence	S50, S50
Risks	S39, S39
hidden / invisible work	S29, S47

Keyword	Source
incompleteness of system or artifact	S50, S46
“it is” versus “it should be”	S38, S29
Skills	S1
customer/management expectations	S78
Changes	S45
Evolution	S22
Deployment	S22
suboptimal decisions	S44
Assessment	S49
Intention	S15
Awarness	S15
Uncertainty	S29

Synonym occurrence

Additionally, several synonyms of TD were found to all definitions that were retrieved from scientific papers. It was done according to procedure described in methodology section. The Table 2 represents occurrence of all synonyms in selected works. Six words or phrases that mean exactly or nearly the same as TD metaphor were found.

Table 2. TD synonyms.

Synonym	Source
Shortcut	S11, S12, S2, S1, S20, S8, S84, S83, S81, S82, S47, S49, S52, S40, S24, S32, S42, S33, S43, S27, S22, S64, S37, S72, S23, S50, S35, S68, S41, S85, S63, S79, S73
“code smells” / design principles violation	S6, S2, S14, S18, S19, S8, S83, S81, S47, S46, S42, S45, S43, S27, S22, S51, S64, S37, S72, S39, S68, S53, S41, S86, S65, S57, S79, S73, S61, S62
workaround / hack	S1, S20, S82, S52, S46, S45, S35, S57
Grime	S81, S46, S45, S22, S53, S57, S62
software aging	S2, S4, S81, S82, S47, S45, S51, S64, S31, S34, S35, S41, S65
spaghetti code	S42, S51, S41

Technical debt types

Apart from defining TD in general, it was found that there are several different types of TD. In order to count each occurrence, every time that some new type of TD was found, it was noted and then used as a search string for automated search process. This automated

search was supposed to ensure the validity of obtained frequency statistics. This procedure is described in more detail in methodology section above. The Table 3 shows every occurrence of TD types in the source works. Overall, seventeen types were distinguished

Table 3. Different TD types with information sources.

Type	Source
design debt	S6, S2, S4, S16, S7, S19, S84, S83, S81, S82, S47, S49, S44, S52, S46, S45, S43, S27, S71, S22, S64, S37, S72, S50, S39, S35, S26, S68, S53, S41, S86, S55, S57, S73, S62
documentation debt	S17, S2, S7, S84, S82, S47, S46, S42, S45, S27, S22, S64, S50, S39, S68, S41, S86, S65, S57
testing debt / test debt / test automation debt	S2, S7, S82, S47, S52, S46, S45, S27, S64, S50, S86, S55, S57, S56, S13, S84, S81, S22, S65
architectural debt	S7, S82, S49, S41, S57, S12, S16, S13, S14, S84, S47, S27, S64, S85, S66, S65, S54
defect debt	S7, S84, S47, S46, S45, S27, S22, S30, S50, S35, S86, S57
build debt	S12, S17, S84, S81, S43, S22, S72, S73
requirements debt	S17, S84, S22, S64, S39
code debt / coding debt	S84, S82, S45, S22, S57
infrastructure debt	S81, S47, S45, S22, S57
people debt	S13, S84, S47, S22, S65
process debt	S12, S27, S22

Type	Source
configuration management debt	S39
social debt	S77
service debt	S22
model debt	S36
technology debt	S31

Keyword, synonyms and TD types map

In order to visualize the frequency of occurrence of each keyword the presented below map was drawn. It is depicted in Figure 10. The more often a keyword on this map occurs in selected definitions the bigger is the node which represents this keyword.

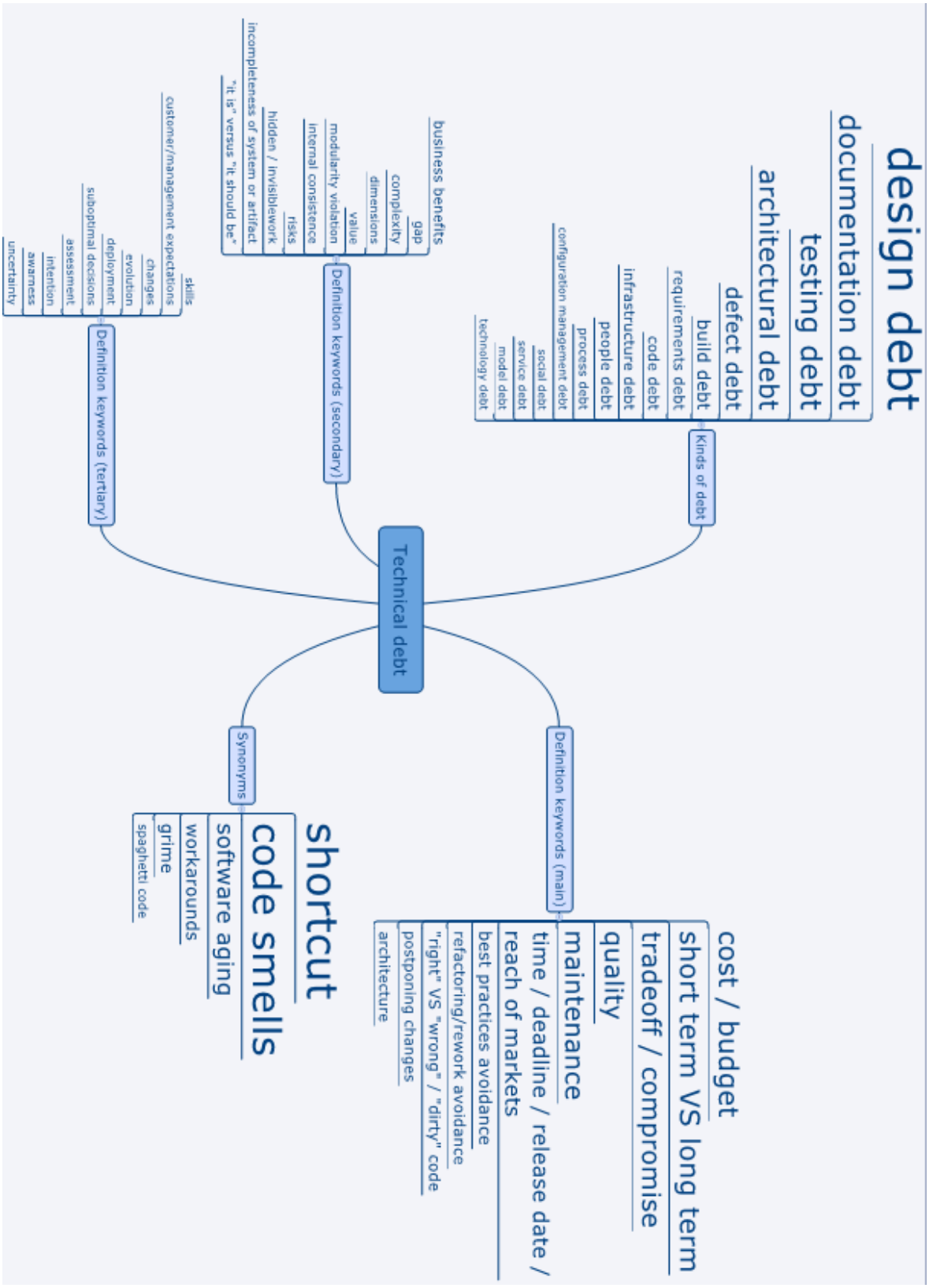

Fig. 10. Keywords map.

5. DISCUSSION

RQ1: What technical debt is?

To answer this question it is necessary to consider the words obtained during the mapping study. All of them can be seen in Table 1 above with relations to the paper which they are originally obtained from. In order to understand the context in which they were used it became necessary to conduct an additional automated search through all works participated in the study. Thus, in this part of the discussion, we will go over all of the most frequently used words in the definition of TD and see their role in the definition of this metaphor. For the better understanding, relationships between the keywords will also be examined and all major conclusions will be drawn in this part of the chapter.

Cost

Among the other keywords the most frequently occurring were the words related to the cost of software development, such as budget, cost, expenses and interest [1], [9], [47], [11]. The words from this set described the short term saved money as well as the necessary extra costs in the long term. In the case when these words were used in sense of savings, the most common purpose of their use was an excuse of accumulating TD. In other words, it is the reason why TD has appeared in this particular project. The both meanings are described by Allman [1] in his definition of TD: "*technical debt is all the shortcuts that save money or speed up progress today at the risk of potentially costing money or slowing down progress in the (usually unclear) future*". The words in the first part of this definition were primarily related to the short-term benefits as well as other advantage descriptions, such as the rapid meeting of market needs, various business benefits or faster release date. Also, a relationship can usually be noticed to those words that describe something that was sacrificed for these benefits. It is necessary to mention here the lack or complete absence of refactoring, modularity violation, and incompleteness of the system or separate artifacts such as documentation, architectural or requirements artifact. It is much easier to write code without taking care of quality but after a while the money which was saved in this way will have to be returned.

On the other hand, cost description of the project has also been used to represent a negative component of TD. If you are going in a debt, you will have to return it sooner or later. Moreover, the later the debt is returned the greater amount of interest has to be paid together with the principal payback. The increase of what is necessary to be returned over time, or in other words, interest is probably the main negative characteristic typical for TD, which motivates its return as soon as it is possible. In this context, the words about the cost, budget and expenses were linked with keywords denoting negative effects for further development and modification of the software system. As the basic concept, quality of the software system can be given as an example. There are also several other keywords related to it such as maintainability of the system, which is deteriorating the more in debt we are. Increasing complexity of the system, which can become eventually deadly for any software project, and additional hidden work, which has to be done before interest of the debt become unbearable, can be other examples of possible negative effects here.

Every point mentioned above increases the cost of system development. The budget allocated to the project can eventually become no longer sufficient despite the initial savings that became possible owing to going into TD.

Quality

Another key issue which has been often mentioned in connection with TD is quality of the software system. The concept of quality can be divided into quality which is visible to the end user and quality which is visible to developers only [37]. Despite the fact that the final system can have really high quality from a user perspective (good performance, high responsiveness, the lack of obvious defects), the situation may be very different as soon as there is a need to look under the hood.

Often much more developer's time is spent on reading code than on writing it [49]. It happens so because in order to add something new or to modify existing code it is necessary to bind it with the existing modules. Here, code convention compliance, code modularity, duplication rate, and code comments coverage characteristics are coming to the fore [26].

Some characteristics of quality code may affect both the quality visible for users and internal quality of the code. Among them unit-tests coverage is worth mentioning. In order to understand the relationship between quality and TD metaphor it is worth looking at each of these characteristics separately.

Code convention compliance determines to what extent the code which is written by one programmer will be readable for another one. It is not a secret that many software projects take place in a team. To add something new to the code which is already written a team member has to read the existing code to understand how it works [62]. To simplify the reading process a large set of rules is usually used. These rules are supposed to govern the naming of variables, the largest possible size of the separate functions or the order of indentation and whitespace usage. Additionally, it must be said that these rules are always under the syntactic rules of the used programming language and can not contradict them. The purpose of these rules is not making the code work properly, but its readability and understandability for the rest of the team [62].

Code duplication leads to the need of making changes to multiple parts of the program when it could be done by a single atomic action. Thus duplication can lead to a loss of some necessary changes and through it to a mismatch in the program, which in turn is likely to become an error. Also, duplication usually increases the complexity of code, because each team member has to remember all the places of its appearance and understand the long chain of actions necessary to modify it [30].

The complexity of code directly affects the time which is required for its understanding and, thus, the cost of modifications. This characteristic is a result of duplication and neglecting the set of rules and conventions described above [41].

Commenting code can degrade its readability in case of its redundancy as well as in case of its deficiency [12]. Each comment should be justified. In other words, comments should be left only where they are needed, but their use in the places which are difficult to understand is extremely desirable. Sometimes, in accordance with the team code convention

comments may be replaced with another similar documentation process. However, ignoring the process at all is likely to lead to the TD of documentation [12].

The unit tests coverage allows a developer to make sure that no changes will harm the existing functionality [46]. Thereby the lack of adequate coverage will lead to increased complexity of introducing changes and potential errors. Correction of these errors as well as larger amount of time required for introducing changes can be characterized as the interests of testing TD. Test driven development (TDD) approach suggests writing unit tests before actual code in order to avoid TD [46].

Complying with all the points mentioned above demands both time and money, which leads to the temptation of ignoring them. It is especially so if an action which supposed to improve the existing situation does not give any immediate visible effect such as profit or saving of costs.

However, neglecting one or more of these characteristics may result in going into TD. The elimination of them will only be possible after some work on refactoring of existing code or completion of missing artifacts such as documentation or unit tests are done.

Time

Time as a characteristic from the selected works is often mentioned due to its shortcoming. According to several authors [7], [67], time is one of the reasons of going into TD. The same need can be expressed in different words such as release date, deadline or reach of markets [1], [43], [63] used in the context of strict requirement which limits the project. Indeed, sometimes a team can face a choice between the complete failure of a project, because impossibility to meet the customer's deadline requirements and sacrificing code quality, which is usually visible only to developers. The second choice here is likely to be meaningful. Acquired in such way, reckless and deliberate (in Fowler’s terms [21]) TD can be paid immediately after the deadline, for example in case when the project follows an incremental or spiral process model and the development team is simply entering a new iteration. However, problems can emerge when software development is carried out within a constant lack of time and as a result it can be followed by infinitely increasing TD.

Developers just are not able to solve the problems created during the previous iterations because the list of new features required for implementation in the current iteration is even bigger than the previous ones. A possible reason of these problems may be some management's misunderstanding of the necessity of the aforementioned procedures that keep internal quality of the product high. Only external quality is taken into account in such case, because it is visible to users and potential buyers of the product.

The strength of the metaphor proposed by Cunnigham consists precisely of its ability to explain the following disastrous consequences to people whose technical expertise is not sufficient. It can prove an importunacy of considering the internal quality improvement of the software as an integral part of any development.

To sum up, we can say that the cost and quality are one of the most important key words describing the TD, whereas the balance between them is the foundation of the whole TD metaphor. Wherein, the representation of time is rather complementary, that is, the time seems to be an important restrictor which can justify TD emergence.

Short term benefits versus long term stability

We have found several mentions of contradistinction between short-term benefits and long-term stability of the project which is sacrificed because of going into TD. Most of the sources agree that one of the most important points is the ability to manage TD and benefit from it in short term [35], [52], [55]. In case of a sharp necessity, for instance lack of time, TD can be used as a provider of resources. On the other hand, constant monitoring and well-timed discontinuation of the resulting software deterioration is essential in order to form long-term stability and reduce the current TD through refactoring procedures or elaborating insufficient or even absent artifacts.

It is often emphasized that TD itself does not have a negative connotation [17], [35], [37]. It can be considered an instrument which is like cash loan. To get benefit from it you have to know how to use it correctly and follow the rules. Problems arise when the mechanisms that generate TD are either ignored or deliberately neglected to obtain greater benefits

without any regard to the enormous challenges awaiting the project in the future. To understand the concept of contrast between short benefits and long term stability it is better to visualize it into example graph.

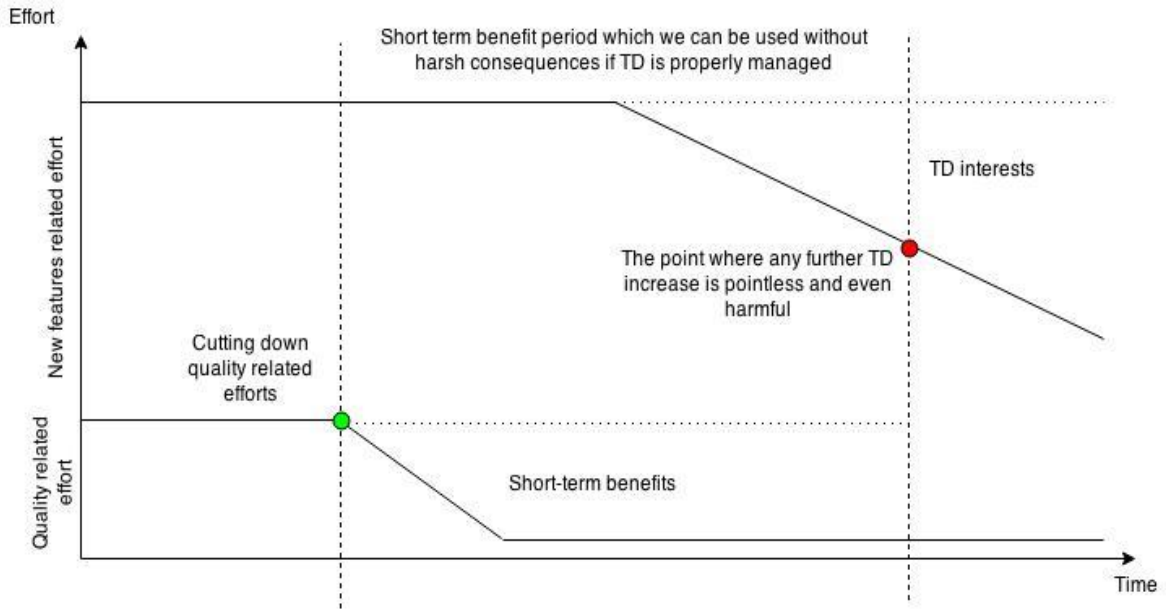

Fig. 11. TD explanation.

On the Figure 11 we can see the hypothetical situation depicted in two dimensions: time and effort. At the moment of beginning we have a level of effort that our team spends on the project, which we can present as an action committed with the intention to make overall progress in the development. On the other hand, we have maintenance activities associated with the project. Their purpose is keeping an appropriate level of produced artifact's quality (the above-mentioned refactoring activities, writing documentation, unit tests, and maintain code in a clean condition with no "code smells"). At the moment when we come to the green point we are in balance. Then, for the sake of short-term benefits, such as for instance finishing the current iteration faster, we are going into TD reducing our efforts spent on quality related activities to some minimum or even abandon them completely. Now, for a while, we can spend our resources only on the improvements that will visible for the user.

The problems appear when we need to turn back to the previously written "dirty code". The less effort on the quality we were spending before the worse the effectiveness of its modification will be. And with the course of time, our efficiency will only be degrading, if we do nothing to change this situation. This is the interest of TD [1].

The red dot is important here. After it, our efficiency is reduced so much that the interest paid on the debt will not allow us to obtain short-term advantage anymore. Starting from here, TD burdens us and sooner or later it may lead to the situation when the code is completely unmaintainable [1], or the further debt increase is not economically feasible anymore.

The way out of this situation would be a review of old code and an attempt to simplify it, to change it in such way that it is becomes easier to read. If it is not a code debt, it is necessary to complete artifacts and perform activities related to this kind of TD, for instance pay some attention to a scanty documentation [2]. In this case, the reasons for wasting efforts will be eliminated and the project will continue to develop in normal pace. The discipline of TD management (TDM) is responsible of determining the best possible pattern of transition between these phases [24]. This activity also involves the measurement of TD and establishing the necessary operations for its repayment.

The TD case which is presented on the graph above is very schematical. The lines would not be so straight in case of real practice. The distance between the green and the red dots as well as the dynamics of interest growth can also vary greatly. However, this diagram explains the essence of metaphor in respect of long-term and short-term decisions for a common understanding of it.

Tradeoff and compromise

Multiple sources report that from a practical point of view TD is a tradeoff or compromise between different points [44], [35], [22]. There have been found different definitions which prove that tradeoff or compromise is an essence of TD. For example Lim et al. defines TD as "*tradeoff among quality, time, and cost*" [44]. Klinger et al. defines it as "*tradeoff between implementing some piece of software in a robust and mature way (the*

"right" way) and taking a shortcut which may provide short term benefits" [35] and Fraser et al. describes it as tradeoff between most appropriate versus immature product: "*trade-offs between delivering the most appropriate –albeit likely immature –product, in the shortest time possible*" [22]. All these definitions compare different pairs of software project characteristics and define TD as a tradeoff between them.

Best practices avoidance and dirty code

Several sources describe the TD problems as committed inadvertently, by carelessness or due to lack of a sufficient level of expertise obtained by personnel engaged in the development [5]. Excessive exaggeration of any mistakes or wrong decisions to TD can overextend metaphor. But still, the inadequate training, lack of right skills, and work on the code without using the best practices adopted from the current industry realities may lead to problems that can be described as going into unintentional TD [65].

There are also other sources of such unintentional TD. As it is described by Klinger et al., "*unintentional debt that the architects and other stakeholders did not actively incur, but which was caused by situations such as acquisition, new alignment requirements, or changes in the market ecosystem*" [35]. Here TD is described as something that resulted from the impact of external factors beyond the control of the development team. External changes were ignored, the project was not adapted, additional resources, such as time and cost, have not been allocated in order to coping with the new reality. Following the basic idea of TD, situation will deteriorate until the debt will not be paid back. The more time is needed for realization of the presence of such debt, the greater the extra effort it will be necessary to spent.

Also TD cannot be simplified or reduced to "dirty code" solely [65], which in fact prevents including "dirty code" keyword in a group of synonyms. However, this phrase occurs in the collected definitions of TD six times, which suggests that it is important. This part is a reference to one of the types of TD, in particular code debt, which will be discussed later.

Postponing changes

Postponing necessary changes can cause TD [63]. However, some sources distinguish the difference between delaying introduction of new functionality and shelving some qualitative changes that are not visible to the end user directly but affect the product stability, as well as its maintainability. According to Kruchten et al., "*New features visible to the user in the form of additional functionality should not be considered as technical debt, as this would also dilute the term*" [37]. On the other hand it is better to differentiate from the situation about which the original Cunnighams definition says: "*"not quite right code" which we postpone making it right"* [8]. Here, the "not quite right code" demands some urgent attention and postponing such kind of changes creates TD.

Maintenance

Maintenance in software engineering is an activity which consists of different actions such as correcting faults, extending system or improving some characteristics of it [53]. The relationship between the maintenance and TD can be expressed as increasing time and effort when significant TD exists. Thus, it can be said that if there is no need to maintain code after writing it, for instance in case of a temporary prototype, the code can be quite "dirty" without compromising the whole project. The problem of TD appears when we need to address to the written code more than once and addressing the code second time and so forth is usually related to the term maintenance [53].

Secondary and tertiary keywords

As it is described in methodology chapter, we consider keywords that occurred five or less times secondary and keywords that occurred only once - tertiary. Low frequency of occurrence of these words can be interpreted as an indication that their use was situational (in the case of secondary keywords) or random (more likely, in the case of tertiary keywords).

Several of these keywords can be used to describe TD alone but due to a sharp decrease in quality of such definitions we decided nevertheless to classify them to keywords. For instance hidden or invisible work may be a variant of unintentional TD but it should not be

considered a full description of TD because it reduces metaphor description to unintentional debt only. Holvitie [29] describes TD as "*The process of accumulating hidden work*", which implies that TD was acquired through ignorance. Also, invisibility can be used as adjective for risks, "*label and communicate (internal) software quality issues, its hidden risks for future development and maintenance as well as their costs incurred*" [47]. This means the lack of understanding of TD mechanisms and severe consequences following absence of TDM.

Seaman and Zazworka define TD as "*Incomplete, immature, or inadequate artifact in the software development lifecycle*" [60]. According to these authors, TD is "*tasks that were left undone, but that run a risk of causing future problems if not completed*" [60]. However incompleteness keyword itself without other characteristics inherent to TD does not provide any information about such required parameters as reasons, consequences and related problems.

Rubin et al. define TD as "*the toll that suboptimal decisions or actions impose on the future welfare of that project*" [57], which raises a new key phrase "suboptimal decisions". Although it may be used alone to describe TD, it would be rather incomplete definition. The mentioned above cite emphasizes that TD is not the suboptimal decision itself but its consequence, "the toll" that need to be paid in due time.

RQ2: Which synonyms of the technical debt metaphor are there?

To answer this question we considered all keywords and phrases that were used in selected papers with the same or similar meaning to TD. This can be important in order to understand TD itself and distinguish it from other concepts which are close but have some important differences.

Bad code smell

The term “bad code smell” (or shortly “code smells” or just “smells”) was introduced by Beck and gained popularity through Fowler's and Beck's book about refactoring process [20]. “Code smells” is basically any symptom in program source code that indicates deeply hidden issues in code or is “*violation of fundamental design principles and negatively impact design quality*” itself. Thus, we can conclude that Cunnigham's TD and Beck's “code smells” terms are closely related. By and large, violations of design principles and coding best practices are TD itself [10], which will slow down the development in the future, draw additional attention and cause wasted efforts. Also, it can be so that “code smells” indicates possible problems hidden in the code. The close relationship and similarity of these two metaphors can be demonstrated by frequent use of metaphors “code smells” in the works referring to Cunnigham (32 out of 84). Examples of “code smells” are:

- Duplicated code in different places of the system.
- Function or procedure which became too big or complex (too many branches and/or loops can indicate the function should be split).
- God object, which performs more than one role and grown up to epic proportions.
- Too many parameters needed to call a function.
- Feature envy: a class that uses some parts of another class too often.
- Excessively long or short identifiers that make the code less readable [68].

Schumacher el al. [58] report about possible automation of “code smells” search. This is applicable for some of its types (particularly “God object” was discussed) detection, which can be very helpful in managing TD process in order to understand the extent of existing problem. Also, we can assess how much effort is needed to alleviate this situation.

Additionally, the term "spaghetti code" has also been used quite often as an indication of complex code. This one has mostly been used to address code debt. Tufano et al. [66] present it as a kind of “code smells” and gives such description: “*Spaghetti Code is a class without structure that declares long methods without parameters*”. Although, according to other sources, this metaphor rather describes code which is confusing to such extent that it

cannot be understood. The flow of its control is difficult to understand, and follow [19]. According to Tufano et al. [66], "code smells" can have side effects such as increase in change- and fault-proneness, or decrease of software understandability and maintainability.

Shortcut

Shortcut is another synonym of TD. It has been widely used in selected papers (33 out of 84) sometimes to describe TD and sometimes as a separate definition of the same issue that the TD metaphor represents. For instance, Brown et al. [3] defines design shortcut in the similar way that TD is defined.

Design shortcuts can give the perception of success until their consequences start slowing projects down. Yli-Huumo et al. [67] defines shortcut as omitted quality, which coincides with the understanding of the TD described in paragraph about short-term benefits versus long-germ stability. According to the same source, "*shortcut, is used to save development time and deliver a solution faster to the customer*" [67], which is relevant to the previously mentioned material in the section about time. Additionally, shortcut itself can be taken in any phase of software development. This corresponds with the different kinds of TD that are described below.

Shull et al. [60] goes even deeper into financial part of TD metaphor and divide the intentional TD into short-term debt and long-term debt, which represent respectively small shortcuts like credit card debt, and strategic actions like a mortgage.

All these points suggest that TD which can be a result of shortcut is deliberate and is made intentionally, unlike unintentional going into TD, which usually happens because a lack of necessary skills or human factor (inattention or error).

Workaround or hack

Workaround word is rarely used to describe TD (8 references from 84 works), but it is enough to include this word in the list of TD synonyms. Some sources describe workaround as something that we would not use if everything were normal. It is code or design which is not obvious and not easy to understand. It can be used to bypass certain

difficult points aside and sometimes it may already be a result of TD. Thus, the need to use a workaround suggests that we are already paying interests and falling into debt even more. Moreover, workaround is not necessarily taken in the coding phase of software development, there can be design or other kinds of workarounds. This makes its concept much closer to TD. Tom et al. [65] in an interview with the developer ascertain that: "*implementing workarounds for issues in the code can take significant time, discovering that you have to implement a workaround in the code can take longer, and it can be quite difficult to find reference information that can tell you that if your thing has this problem, then you need to do this thing*". On the other hand, Lim [43] argues that the inclusion of workarounds, hacks and band-aid solutions to TD is a desirable extension to the original Cunnigham's metaphor.

Generally, hack or workaround is something that is created to avoid some significant (for instance time consuming) challenge instead of facing it. This approach can lead to problems in the future that can be considered a part of TD.

Grime and Rot

Both concepts are rather less related to TD than previously mentioned ones. Seaman et al. [60] define grime as "*accumulation of non-pattern code in classes following a design pattern*" and rot as "*changes that break the integrity of a design pattern*". So, basically, both concepts describe dilution and subsequent destruction of the code which originally followed certain patterns. These patterns were supposed to make code more readable and maintainable but the dilution of them with non-paternal parts can become TD in the future.

Software aging

Software aging concept implies that software can degrade even if nobody has touched it since it is developed. Falessi et al. [17] claim that "*systems to remain useful must change, and that change will increase their complexity, leading to software decay if refactoring is not done as needed*". This is different from the concept of TD in a sense that no symptoms indicating the presence of a TD appear over time in the system. However, the longer the system remains unchanged the less it is needed by the end user. Thus, to avoid system aging the constant modification of the whole system is needed.

Modification in turn may cause TD if it is made without employing best practices available.

RQ3: What types of technical debt are mentioned in scientific works?

As it is mentioned before, TD cannot be reduced to just "dirty code". Problems that are similar in characteristics described with Cunnigham's metaphor may arise in other areas related to software development. All these problems may have similar dynamics or common properties. Also, all of them have in common that their disregard for a long time can cause pernicious effects [2].

In order to find the answer to this research question, mention of various TD types were collected from our scientific papers sample. As a result, the classified core group consists of TD types that arise at most software lifecycle stages and during different related to the lifecycle activities.

The most frequently mentioned of them in the selected works were:

- Requirements debt refers to incomplete requirements implementation [2].
- Architectural debt refers to problems in software architecture or possibly to architecturally significant requirements. This debt usually cannot be paid by simple correction of code because roots of existing problems are deeper [2], [36], [65].
- Design debt refers to TD incurred because of good object-oriented design principles violation [2], [25], [51], [65].
- Code debt refers to problems in software code which make it difficult to read and maintain. Usually it is related to bad coding practices and can be eliminated by refactoring [2], [51], [65].
- Testing debt, test debt, and test automation debt refer to debt appeared because of insufficient testing or test automation activities performed [2].
- Configuration management debt refers to debt acquired because deficiencies in eponymous activity [47].
- Documentation debt refers to missing, inadequate or incomplete documentation related to the project [2] [25], [65].

In addition to this group some authors have examined and proposed in their works some additional types described below.

Social debt and people debt

Several authors claim that the TD metaphor can be transferred from purely technical field in the social one because they have been always interrelated with each other. According to Tamburri et al. [64], "Social debt" in software engineering informally refers to unforeseen project cost connected to a "suboptimal developed community" or the degree to which it is immature or unable to tackle a certain development problem. This concept can present lack of trust within a community, which can lead to project failure. But the authors [64] admit that this concept is vague and "*far from being a clear-cut financial figure*", which is quite opposite to the TD concept.

Six other papers suggest that there is people debt. An example of it is given by Alves et al. [2]. It is when expertise is concentrated in too few people, as an effect of delayed training and/or hiring. This situation can affect overall stability of the project and make it dependant from decisions of few people. Disappearance of them can cause disastrous consequences for the whole project.

Model debt

According to Lindgren [45], model debt measures and communicates the software quality of the domain model. It consists of three parts:

- Business isolation debt. It indicates how well the business logic is isolated from technical infrastructure and irrelevant business logic.
- Business behavior debt. It indicates the amount of knowledge crunching (process of distilling of the complex and unstructured knowledge of the domain experts into rigorous concepts which form a domain model) carried out in the domain model.
- Inefficient business debt. It indicates how efficiently the current domain model solves problems in the domain.

Technology debt

According to Magnusson and Widel, technology debt is defined as "*Accumulated obligation owned by current CIO (debtor) to future CIO (creditor), where previous decisions limit prospective decisions*" [32]. In general, this concept is rather a derivate from TD than is its kind.

Defect debt

According to Seaman [59], defect debt means knowing about existing defects and still postponing fixing them. Several authors argue with this approach to TD claiming that the addition of defects in the concept of TD metaphor only dilutes it [37].

Build debt

Alves et al. [2] states that build debt refers to build activity and related issues that make this task harder or more time/processing consuming without any purpose. There can be unnecessary code or ill-defined dependencies which should be deleted instead of keeping them inside the project. Without doing this, the building process becomes slower. It is more difficult to integrate project continuously with significant build debt accrued.

Infrastructure and automation debt

Infrastructure debt refers to infrastructure issues that can delay or hinder some development activities if they present in the software organization [2]. Some examples of this debt kind are delaying an upgrade or postponing infrastructure fix. Codabux et al. define it as "*work that improves the team's process and ability to produce a quality product*" [7], which is quite similar to automation debt concept. Automation debt is defined as "*the work involved in automating tests of previously developed functionality to support continuous integration and faster development cycles*" [7].

Process debt

Process debt indicates that the process which is employed by developers is outdated and needs renovation. The process could be designed to handle issues that are not existed anymore. Also, it may have a lack of embedded tools which would be able to help developers cope with difficulties that they face regularly. It is known that process itself is continuously developing together with the software system [18]. Paying less attention than it is needed to process-related problems can cause this kind of debt [37], [2], [27].

Service debt

Service debt is briefly mentioned by Alves et al. [2] as a type of TD when there is a need of service substitution driven by business or technical reasons. Without well-timed replacement or renewal services can become tedious to maintain and cause distraction to developers especially in case of large service ecosystem. There are several dimensions that can be covered by service debt metaphor: selection, composition and operation of the services.

RQ4: Is it possible to build the general definition of technical debt metaphor?

In order to summarize everything discussed above, we tried to gather all the keywords retrieved from selected articles in definitions of TD. Construction of the final definitions, which are supposed to incorporate all important keywords, is the last stage of this research. Due to big quantity of keywords it was difficult to incorporate them in one big definition because the definition of such size does not make much sense. Therefore, we decided to create several descriptions of TD from the different points of view.

Building the definitions we actively used keywords frequency of occurrence statistic, which is one of the results of this study. In order to build the definitions all the most used keywords were combined together in such a manner that they emphasize the main contradistinctions discussed above: short term benefit and long term stability, quality, time

and cost categories, and postponing changes instead of employing best practices. All three definitions are provided below:

> A compromise between short term benefit (for example meeting of immediate market needs) and long term stability, extensibility and internal consistence of the system under development.
>
> A tradeoff among quality, time and cost of the system which is made in favor of making shortcut that eventually leads to increased complexity of the system and quality gap between current situation and what should have been otherwise.
>
> A decision to postpone necessary changes or to do them "fast and dirty" and pay for that with future maintenance cost instead of employing best practices right away.

The proposed definitions cannot describe TD metaphor in its full complexity. They rather aim to summarize the whole discussion made above and give a short overview of all knowledge gathered during this research.

There is also another way to define TD. Instead of defining the very concept, we can distinguish what TD is not and thus, limit its definition or in other words, put it in a frame that will determine TD better than a direct description. Most of such frames were outlined in the discussion above by mentioning them in conjunction with the relevant keywords. So, as it is mentioned above, TD should not be simplified to the level of one of the types described above, such as code debt for example. Also, TD should not be equated with defects or errors in software project because it can dilute the definition.

6. CONCLUSION

TD is a widespread metaphor which describes the problem of avalanche-like increasing complexity of software project, deterioration of its maintainability and prospects for further development in the case of neglecting necessary work related to internal quality. In recent years, more and more practitioners have been using this metaphor to explain existing problems in their software project. Therefore it is important to understand the meaning of this metaphor to pinpoint and solve the existing problem. In order to do that systematic mapping study was performed. We finished this study by creating keyword map of the most frequently used words and phrases that describe TD. It is also important to remember that TD is not only one metaphor which aims to describe problems of software deterioration. In order to understand this problem better, all derivates and synonyms of TD were gathered and classified. Also, TD can be divided into types. This division allows allocating and systematizing distinctions that are emphasized by various sources which were considered during the study. Overall, 33 keywords or phrases were found, 6 synonyms and 17 of TD were distinguished.

Despite all attempts that were made in defining TD, there is no simple answer what TD is. This phenomenon is complex and strongly dependent on the realities of the particular project. Therefore it is rather matter for discussion than a strong statement.

Despite existing secondary studies [42] and various interview with practitioners [43] enabling to define the problem of TD in software engineering better, the topic still has much room for additional research activities which hopefully might be based on the results of this work.

REFERENCES

1. ALLMAN, E., 2012. Managing Technical Debt. *Commun.ACM*, **55**(5), pp. 50-55.
2. ALVES, N.S., RIBEIRO, L.F., CAIRES, V., MENDES, T.S. and SPÍNOLA, R.O., 2014. Towards an Ontology of Terms on Technical Debt, *Managing Technical Debt (MTD), 2014 Sixth International Workshop on* 2014, IEEE, pp. 1-7.
3. BROWN, N., CAI, Y., GUO, Y., KAZMAN, R., KIM, M., KRUCHTEN, P., LIM, E., MACCORMACK, A., NORD, R., OZKAYA, I., SANGWAN, R., SEAMAN, C., SULLIVAN, K. and ZAZWORKA, N., 2010. Managing Technical Debt in Software-reliant Systems, *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* 2010, ACM, pp. 47-52.
4. BUDGEN, D., KITCHENHAM, B., CHARTERS, S., TURNER, M., BRERETON, P. and LINKMAN, S., 2007. Preliminary results of a study of the completeness and clarity of structured abstracts, Proc. of the 11th Int. Conf. on Evaluation and Assessment in Software Engineering 2007, pp. 64-72.
5. C. SEAMAN, R. O. SPINOLA, "Managing Technical Debt," [Short Course] XVII Brazilian Symposium on Software Quality, Salvador, Brazil, 2013
6. CAST, 2011. Reduce Technical Debt, Available from: http://www.castsoftware.com/solutions/reduce-technical-debt (accessed 19.10.11).
7. CODABUX, Z. and WILLIAMS, B., 2013. Managing technical debt: An industrial case study, *Managing Technical Debt (MTD), 2013 4th International Workshop on* 2013, pp. 8-15.
8. CUNNINGHAM, W., 1992. The WyCash Portfolio Management System. SIGPLAN OOPS Mess., 4(2), pp. 29-30.
9. CURTIS, B., 2012. Paying Down the Interest on Your Applications.
10. CURTIS, B., SAPPIDI, J. and SZYNKARSKI, A., 2012. Estimating the Principal of an Application's Technical Debt. *Software, IEEE*, **29**(6), pp. 34-42.
11. DE GROOT, J., NUGROHO, A., BACK, T. and VISSER, J., 2012. What is the value of your software? *Managing Technical Debt (MTD), 2012 Third International Workshop on* 2012, pp. 37-44.
12. DEPASQUALE, P.J., LOCASTO, M.E. and KACZMARCZYK, L.C., 2012. Identifying Effective Pedagogical Practices for Commenting Computer Source

Code (Abstract Only), *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* 2012, ACM, pp. 678-678.

13. DIXON-WOODS, M., AGARWAL, S., JONES, D., YOUNG, B. & SUTTON, A. (2005), 'Synthesising qualitative and quantitative evidence: a review of possible methods.', *Journal of Health Services Research and Policy* 10(1), 45–53.
14. DYBA, T., KAMPENES, V.B. and SJOBERG, D.I.K. A Systematic Review of Statistical Power in Software Engineering Experiments, Information and Software Technology, 48(8): 745–755, 2006
15. EL EMAM, K., KORU, A.G., 2008. A Replicated Survey of IT Software Project Failures. *Software, IEEE,* **25**(5), pp. 84-90.
16. ELM, J., 2009. Design Debt Economics: A Vocabulary for Describing the Causes, Costs and Cures for Software Maintainability Problems. IBM, Available: http://www.ibm.com/developerworks/rational/library/edge/09/jun09/designdebteconomics/ (accessed 24.08.11).
17. FALESSI, D., KRUCHTEN, P., NORD, R.L. and OZKAYA, I., 2014. Technical Debt at the Crossroads of Research and Practice: Report on the Fifth International Workshop on Managing Technical Debt. *SIGSOFT Softw.Eng.Notes,***39**(2), pp. 31-33.
18. FEILER, P.H. and HUMPHREY, W.S., 1993. Software process development and enactment: concepts and definitions, *Software Process, 1993. Continuous Software Process Improvement, Second International Conference on the*1993, pp. 28-40.
19. FOOTE, B. and YODER, J., 1997. Big ball of mud. *Pattern languages of program design,* **4**, pp. 654-692.
20. FOWLER, M., BECK, K. 1999. Refactoring: improving the design of existing code. Addison Wesley.
21. FOWLER, M., Technical debt quadrant. Available at: www.martinfowler.com/bliki/TechnicalDebtQuadrant.html, 2009. Last visited 08.04.2015.
22. FRASER, S., MANCL, D., OPDYKE, B., BISHOP, J., KATHAIL, P., LACAR, J., OZKAYA, I. and SZYNKARSKI, A., 2013. Technical Debt: From Source to Mitigation, *Proceedings of the 2013 Companion Publication for Conference on*

Systems, Programming, \&\#38; Applications: Software for Humanity 2013, ACM, pp. 67-70.

23. GOLDEN J.M., Transformation patterns for curing the human causes of technical debt, Cutter IT Journal 23 (10) (2010) 30–35.
24. GRIFFITH, I., IZURIETA, C., TAFFAHI, H. and CLAUDIO, D., 2014. A Simulation Study of Practical Methods for Technical Debt Management in Agile Software Development, *Proceedings of the 2014 Winter Simulation Conference* 2014, IEEE Press, pp. 1014-1025.
25. GUO, Y. and SEAMAN, C., 2011. A Portfolio Approach to Technical Debt Management, *Proceedings of the 2Nd Workshop on Managing Technical Debt* 2011, ACM, pp. 31-34.
26. GUO, Y., SEAMAN, C., ZAZWORKA, N. and SHULL, F., 2010. Domain-specific Tailoring of Code Smells: An Empirical Study, *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*2010, ACM, pp. 167-170.
27. GUO, Y., SPÍNOLA, R.O. and SEAMAN, C., 2014. Exploring the costs of technical debt management–a case study. *Empirical Software Engineering,* , pp. 1-24.
28. HANNAY, J., SJOBERG, D.I.K. and DYBA, T. A Systematic Review of Theory Use in Software Engineering Experiments, IEEE Transactions on Software Engineering, 33(2): 87-107, 2007
29. HOLVITIE, J., 2014. Software implementation knowledge management with technical debt and network analysis, *Research Challenges in Information Science (RCIS), 2014 IEEE Eighth International Conference on* 2014, IEEE, pp. 1-6.
30. HORDIJK, W.T.B., PONISIO, M.L. and WIERINGA, R.J., 2009. Harmfulness of Code Duplication - A Structured Review of the Evidence, *13th International Conference on Evaluation and Assessment in Software Engineering (EASE), Durham University, UK*, April 2009, British Computer Society.
31. *IEEE Standard for Developing Software Life Cycle Processes.* 2006.
32. JOHANSSON, A. and WIDELL, C., 2014. Evaluating future IT-investment options through technology debt-A case study and theory testing approach on Company AB.

33. KEELE, S., 2007. Guidelines for performing systematic literature reviews in software engineering.
34. KITCHENHAM, B., PEARL BRERETON, O., BUDGEN, D., TURNER, M., BAILEY, J. and LINKMAN, S., 2009. Systematic literature reviews in software engineering – A systematic literature review. Information and Software Technology, 51(1), pp. 7-15.
35. KLINGER, T., TARR, P., WAGSTROM, P. and WILLIAMS, C., 2011. An Enterprise Perspective on Technical Debt, *Proceedings of the 2Nd Workshop on Managing Technical Debt* 2011, ACM, pp. 35-38.
36. KRUCHTEN, P., NORD, R.L. and OZKAYA, I., 2012. Technical Debt: From Metaphor to Theory and Practice. *Software, IEEE,* **29**(6), pp. 18-21.
37. KRUCHTEN, P., NORD, R.L., OZKAYA, I. and FALESSI, D., 2013. Technical Debt: Towards a Crisper Definition Report on the 4th International Workshop on Managing Technical Debt. *SIGSOFT Softw.Eng.Notes,* **38**(5), pp. 51-54.
38. KRUCHTEN, P., NORD, R.L., OZKAYA, I. and VISSER, J., "Technical Debt in Software Development: from Metaphor to Theory--Report on the Third International Workshop on Managing Technical Debt, held at ICSE 2012 " *ACM SIGSOFT Software Engineering Notes*, vol. 37(5), pp. 36-38, 2012.
39. KRUCHTEN, P., NORD, R.L. and OZKAYA, I. "Technical Debt: From Metaphor to Theory and Practice," *IEEE Software*, vol. 29(6), pp. 18-21, 2012.
40. KTATA O., LEVESQUE G., 2010. Designing and implementing a measurement program for scrum teams: what do agile developers really need and want? 3rd C* Conference on Computer Science and Software Engineering 2010, C3S2E '10, Montreal, QC, Canada, May 19, 2010–May 20, 2010. Association for Computing Machinery, 101–107, *A5
41. LANNING, D.L. and KHOSHGOFTAAR, T.M., 1994. Modeling the relationship between source code complexity and maintenance difficulty. *Computer,* **27**(9), pp. 35-40.
42. LI, Z., AVGERIOU, P. and LIANG, P., 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software,* **101**(0), pp. 193-220.
43. LIM, E., 2012. Technical debt: what software practitioners have to say.

44. LIM, E., TAKSANDE, N. and SEAMAN, C., 2012. A Balancing Act: What Software Practitioners Have to Say about Technical Debt. *Software, IEEE,* **29**(6), pp. 22-27.
45. LINDGREN, M., 2012. Bridging the software quality gap.
46. MADEYSKI, L., 2010. The impact of Test-First programming on branch coverage and mutation score indicator of unit tests: An experiment. *Information and Software Technology,* **52**(2), pp. 169-184.
47. MAYR, A., PLOSCH, R. and KORNER, C., 2014. A Benchmarking-Based Model for Technical Debt Calculation, *Quality Software (QSIC), 2014 14th International Conference on* 2014, IEEE, pp. 305-314.
48. MCCONNELL S., "Managing technical debt (slides)," in Workshop on Managing Technical Debt (part of ICSE 2013): IEEE, 2013.
49. MCCONNELL, S., 2004. *Code complete.* Microsoft press.
50. MCCONNELL, S., 2008. "Managing Technical Debt." Best Practices White Paper, Construx. —. 2006. Software Estimation: Demystifying the Black Art. Microsoft Press.
51. N. ZAZWORKA, R. O. SPINOLA, A. VETRO, F. SHULL and C. SEAMAN, "A case study on effectively identifying technical debt," *EASE '13: Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, 2013
52. NORD, R.L., OZKAYA, I., KRUCHTEN, P. and GONZALEZ-ROJAS, M., 2012. In search of a metric for managing architectural technical debt, *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on* 2012, IEEE, pp. 91-100.
53. NUGROHO, A., VISSER, J. and KUIPERS, T., 2011. An Empirical Model of Technical Debt and Interest, *Proceedings of the 2Nd Workshop on Managing Technical Debt* 2011, ACM, pp. 1-8.
54. PETERSEN, K., FELDT, R., MUJTABA, S. and MATTSSON, M., 2008. Systematic mapping studies in software engineering, 12th International Conference on Evaluation and Assessment in Software Engineering 2008, sn.
55. RAMASUBBU, N. and KEMERER, C.F., 2013. Managing Technical Debt in Enterprise Software Packages.

56. ROTHMAN, J., 2006. An incremental technique to pay off testing technical debt. Web.http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=COL&ObjectId=11011&tth=DYN&tt=siteemail&iDyn=2. (Accessed 1 May 2008)
57. RUBIN, Y., KALLNER, S., GUY, N. and SHACHOR, G., 2013. Restraining technical debt when developing large-scale Ajax applications, *WEB 2013, The First International Conference on Building and Exploring Web Based Environments* 2013, pp. 13-18.
58. SCHUMACHER, J., ZAZWORKA, N., SHULL, F., SEAMAN, C. and SHAW, M., 2010. Building Empirical Support for Automated Code Smell Detection, *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* 2010, ACM, pp. 8:1-8:10.
59. SEAMAN, C. and GUO, Y., 2011. Measuring and monitoring technical debt. *Advances in Computers*, **82**, pp. 25-46.
60. SEAMAN, C. and ZAZWORKA, N., 2011. Technical Debt.
61. SHULL, F., FALESSI, D., SEAMAN, C., DIEP, M. and LAYMAN, L., 2013. Technical debt: showing the way for better transfer of empirical results. *Perspectives on the Future of Software Engineering.* Springer, pp. 179-190.
62. SMIT, M., GERGEL, B., HOOVER, H.J. and STROULIA, E., 2011. Code convention adherence in evolving software, *Software Maintenance (ICSM), 2011 27th IEEE International Conference on* 2011, pp. 504-507.
63. TAKSANDE, N., 2011. Empirical study on technical debt as viewed by software practitioners. University of Maryland, Baltimore County.
64. TAMBURRI, D.A., KRUCHTEN, P., LAGO, P. and VAN VLIET, H., 2013. What is social debt in software engineering? *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on*2013, pp. 93-96.
65. TOM, E., AURUM, A. and VIDGEN, R., 2013. An exploration of technical debt. *Journal of Systems and Software*, **86**(6), pp. 1498-1516.
66. TUFANO, M., PALOMBA, F., BAVOTA, G., OLIVETO, R., DI PENTA, M., DE LUCIA, A. and POSHYVANYK, D., 2015. When and Why Your Code Starts to Smell Bad.

67. YLI-HUUMO, J., MAGLYAS, A. and SMOLANDER, K., 2014. The Sources and Approaches to Management of Technical Debt: A Case Study of Two Product Lines in a Middle-Size Finnish Software Company. **8892**, pp. 93-107.
68. ZAZWORKA, N., IZURIETA, C., WONG, S., CAI, Y., SEAMAN, C. and SHULL, F., 2013. Comparing four approaches for technical debt identification. *Software Quality Journal,* , pp. 1-24.

APPENDIX 1. List of selected scientific works

I. ALLMAN, E., 2012. Managing Technical Debt. *Commun.ACM*, **55**(5), pp. 50-55.

II. ALVES, N.S., RIBEIRO, L.F., CAIRES, V., MENDES, T.S. and SPÍNOLA, R.O., 2014. Towards an Ontology of Terms on Technical Debt, *Managing Technical Debt (MTD), 2014 Sixth International Workshop on* 2014, IEEE, pp. 1-7.

III. ALZAGHOUL, E. and BAHSOON, R., 2013. CloudMTD: Using real options to manage technical debt in cloud-based service selection, *Managing Technical Debt (MTD), 2013 4th International Workshop on* 2013, pp. 55-62.

IV. ALZAGHOUL, E. and BAHSOON, R., 2013. Economics-Driven Approach for Managing Technical Debt in Cloud-Based Architectures, *Utility and Cloud Computing (UCC), 2013 IEEE/ACM 6th International Conference on* 2013, pp. 239-242.

V. ALZAGHOUL, E. and BAHSOON, R., 2014. Evaluating Technical Debt in Cloud-Based Architectures Using Real Options, *Software Engineering Conference (ASWEC), 2014 23rd Australian* 2014, pp. 1-10.

VI. AMMERLAAN, E., VENINGA, W. and ZAIDMAN, A., 2013. Old Habits Die Hard: Why Refactoring for Understandability Does Not Give Immediate Benefits}, *Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)}* 2013.

VII. BERENBACH, B., 2014. On Technical Credit. *Procedia Computer Science*, **28**(0), pp. 505-512.

VIII. BROWN, N., CAI, Y., GUO, Y., KAZMAN, R., KIM, M., KRUCHTEN, P., LIM, E., MACCORMACK, A., NORD, R., OZKAYA, I., SANGWAN, R., SEAMAN, C., SULLIVAN, K. and ZAZWORKA, N., 2010. Managing Technical Debt in Software-reliant Systems, *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* 2010, ACM, pp. 47-52.

IX. CODABUX, Z. and WILLIAMS, B., 2013. Managing technical debt: An industrial case study, *Managing Technical Debt (MTD), 2013 4th International Workshop on* 2013, pp. 8-15.

X. CURTIS, B., 2012. Paying Down the Interest on Your Applications.

XI. CURTIS, B., SAPPIDI, J. and SZYNKARSKI, A., 2012. Estimating the Principal of an Application's Technical Debt. *Software, IEEE,* **29**(6), pp. 34-42.

XII. DE GROOT, J., NUGROHO, A., BACK, T. and VISSER, J., 2012. What is the value of your software? *Managing Technical Debt (MTD), 2012 Third International Workshop on* 2012, pp. 37-44.

XIII. DOS SANTOS, P., VARELLA, A., DANTAS, C. and BORGES, D., 2013. Visualizing and Managing Technical Debt in Agile Development: An Experience Report. **149**, pp. 121-134.

XIV. EISENBERG, R.J., 2012. A Threshold Based Approach to Technical Debt. *SIGSOFT Softw.Eng.Notes,* **37**(2), pp. 1-6.

XV. FALESSI, D., KRUCHTEN, P., NORD, R.L. and OZKAYA, I., 2014. Technical Debt at the Crossroads of Research and Practice: Report on the Fifth International Workshop on Managing Technical Debt. *SIGSOFT Softw.Eng.Notes,***39**(2), pp. 31-33.

XVI. FERNÁNDEZ SÁNCHEZ, C., DÍAZ FERNÁNDEZ, J., GARBAJOSA SOPEÑA, J. and PÉREZ BENEDÍ, J., 2013. A Cost-Benefit analysis model for technical debt management considering uncertainty and time.

XVII. FRASER, S., MANCL, D., OPDYKE, B., BISHOP, J., KATHAIL, P., LACAR, J., OZKAYA, I. and SZYNKARSKI, A., 2013. Technical Debt: From Source to Mitigation, *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, \&\#38; Applications: Software for Humanity* 2013, ACM, pp. 67-70.

XVIII. GRIFFITH, I., IZURIETA, C., TAFFAHI, H. and CLAUDIO, D., 2014. A Simulation Study of Practical Methods for Technical Debt Management in Agile Software Development, *Proceedings of the 2014 Winter Simulation Conference* 2014, IEEE Press, pp. 1014-1025.

XIX. GRIFFITH, I., REIMANIS, D., IZURIETA, C., CODABUX, Z., DEO, A. and WILLIAMS, B., 2014. The Correspondence between Software Quality Models and Technical Debt Estimation Approaches, *Managing Technical Debt (MTD), 2014 Sixth International Workshop on* 2014, IEEE, pp. 19-26.

XX. GUO, Y. and SEAMAN, C., 2011. A Portfolio Approach to Technical Debt Management, *Proceedings of the 2Nd Workshop on Managing Technical Debt* 2011, ACM, pp. 31-34.

XXI. GUO, Y., SEAMAN, C., ZAZWORKA, N. and SHULL, F., 2010. Domain-specific Tailoring of Code Smells: An Empirical Study, *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*2010, ACM, pp. 167-170.

XXII. GUO, Y., SPÍNOLA, R.O. and SEAMAN, C., 2014. Exploring the costs of technical debt management–a case study. *Empirical Software Engineering*, , pp. 1-24.

XXIII. HO, T.T. and RUHE, G., 2014. When-to-release decisions in consideration of technical debt, *Managing Technical Debt (MTD), 2014 Sixth International Workshop on* 2014, IEEE, pp. 31-34.

XXIV. HOLVITIE, J. and LEPPANEN, V., 2013. DebtFlag: Technical debt management with a development environment integrated tool, *Managing Technical Debt (MTD), 2013 4th International Workshop on* 2013, pp. 20-27.

XXV. HOLVITIE, J., 2014. Software implementation knowledge management with technical debt and network analysis, *Research Challenges in Information Science (RCIS), 2014 IEEE Eighth International Conference on* 2014, IEEE, pp. 1-6.

XXVI. HOLVITIE, J., LEPPANEN, V. and HYRYNSALMI, S., 2014. Technical Debt and the Effect of Agile Software Development Practices on It-An Industry Practitioner Survey, *Managing Technical Debt (MTD), 2014 Sixth International Workshop on* 2014, IEEE, pp. 35-42.

XXVII. IZURIETA, C., GRIFFITH, I., REIMANIS, D. and LUHR, R., 2013. On the Uncertainty of Technical Debt Measurements, *Information Science and Applications (ICISA), 2013 International Conference on* 2013, pp. 1-4.

XXVIII. IZURIETA, C., VETRO, A., ZAZWORKA, N., YUANFANG CAI, SEAMAN, C. and SHULL, F., 2012. Organizing the technical debt landscape, *Managing Technical Debt (MTD), 2012 Third International Workshop on* 2012, pp. 23-26.

XXIX. JOHANSSON, A. and WIDELL, C., 2014. Evaluating future IT-investment options through technology debt-A case study and theory testing approach on Company AB.

XXX. KIM, M., CAI, D. and KIM, S., 2011. An Empirical Investigation into the Role of API-level Refactorings During Software Evolution, *Proceedings of the 33rd International Conference on Software Engineering* 2011, ACM, pp. 151-160.

XXXI. KIM, M., ZIMMERMANN, T. and NAGAPPAN, N., 2012. A Field Study of Refactoring Challenges and Benefits, *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* 2012, ACM, pp. 50:1-50:11.

XXXII. KLINGER, T., TARR, P., WAGSTROM, P. and WILLIAMS, C., 2011. An Enterprise Perspective on Technical Debt, *Proceedings of the 2Nd Workshop on Managing Technical Debt* 2011, ACM, pp. 35-38.

XXXIII. KRISHNA, V. and BASU, A., 2012. Minimizing Technical Debt: Developer's viewpoint, Software Engineering and Mobile Application Modelling and Development (ICSEMA 2012), International Conference on 2012, pp. 1-5.

XXXIV. KRISHNA, V. and BASU, A., 2013. Software Engineering Practices for Minimizing Technical Debt.

XXXV. KRUCHTEN, P., 2012. Strategic Management of Technical Debt: Tutorial Synopsis, *Quality Software (QSIC), 2012 12th International Conference on* 2012, pp. 282-284.

XXXVI. KRUCHTEN, P., NORD, R.L. and OZKAYA, I., 2012. Technical Debt: From Metaphor to Theory and Practice. *Software, IEEE,* **29**(6), pp. 18-21.

XXXVII. KRUCHTEN, P., NORD, R.L., OZKAYA, I. and FALESSI, D., 2013. Technical Debt: Towards a Crisper Definition Report on the 4th International Workshop on Managing Technical Debt. *SIGSOFT Softw.Eng.Notes,* **38**(5), pp. 51-54.

XXXVIII. KRUCHTEN, P., NORD, R.L., OZKAYA, I. and VISSER, J., 2012. Technical Debt in Software Development: From Metaphor to Theory Report on the Third International Workshop on Managing Technical Debt. *SIGSOFT Softw.Eng.Notes*, **37**(5), pp. 36-38.

XXXIX. LANE, J.A., KOOLMANOJWONG, S. and BOEHM, B., 2013. 4.6. 3 Affordable Systems: Balancing the Capability, Schedule, Flexibility, and Technical Debt Tradespace, *INCOSE International Symposium* 2013, Wiley Online Library, pp. 1385-1399.

XL. LASSENIUS, K.R.C., 2014. PACING SOFTWARE PRODUCT DEVELOPMENT: A Framework and Practical Implementation Guidelines.

XLI. LETOUZEY, J. and ILKIEWICZ, M., 2012. Managing Technical Debt with the SQALE Method. *Software, IEEE,* **29**(6), pp. 44-51.

XLII. LETOUZEY, J.-., 2012. The SQALE method for evaluating Technical Debt, *Managing Technical Debt (MTD), 2012 Third International Workshop on* 2012, pp. 31-36.

XLIII. LI, Z., AVGERIOU, P. and LIANG, P., 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software,* **101**(0), pp. 193-220.

XLIV. LI, Z., LIANG, P., AVGERIOU, P., GUELFI, N. and AMPATZOGLOU, A., 2014. An Empirical Investigation of Modularity Metrics for Indicating Architectural Technical Debt, *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures* 2014, ACM, pp. 119-128.

XLV. LIM, E., 2012. Technical debt: what software practitioners have to say.

XLVI. LIM, E., TAKSANDE, N. and SEAMAN, C., 2012. A Balancing Act: What Software Practitioners Have to Say about Technical Debt. *Software, IEEE,* **29**(6), pp. 22-27.

XLVII. LINDGREN, M., 2012. Bridging the software quality gap.

XLVIII. MAMUN, M.A.A., BERGER, C. and HANSSON, J., 2014. Explicating, Understanding, and Managing Technical Debt from Self-Driving Miniature Car Projects, *Managing Technical Debt (MTD), 2014 Sixth International Workshop on* 2014, IEEE, pp. 11-18.

XLIX. MAR, K. and JAMES, M., 2006. Technical Debt and Design Death.

L. MAYR, A., PLOSCH, R. and KORNER, C., 2014. A Benchmarking-Based Model for Technical Debt Calculation, *Quality Software (QSIC), 2014 14th International Conference on* 2014, IEEE, pp. 305-314.

LI. MCGREGOR, J.D., MONTEITH, J.Y. and JIE ZHANG, 2012. Technical debt aggregation in ecosystems, *Managing Technical Debt (MTD), 2012 Third International Workshop on* 2012, pp. 27-30.

LII. MONTEITH, J.Y. and MCGREGOR, J.D., 2013. Exploring software supply chains from a technical debt perspective, *Managing Technical Debt (MTD), 2013 4th International Workshop on* 2013, pp. 32-38.

LIII. MORRISON-SMITH, S., DIGHANS, S., DANIELS, T., MARMON, C. and IZURIETA, C., 2013. Technical Debt Reduction Using a Game Theoretic Competitive Source Control Approach.

LIV. NORD, R.L., OZKAYA, I., KRUCHTEN, P. and GONZALEZ-ROJAS, M., 2012. In search of a metric for managing architectural technical debt, *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on* 2012, IEEE, pp. 91-100.

LV. NORD, R.L., OZKAYA, I., SANGWAN, R.S. and BROWN, N., 2012. 9 Communicating the Benefits of Architecting within Agile Development: Quantifying the Value of Architecting within Agile Software Development via Technical Debt Analysis. *Results of SEI Line-Funded Exploratory New Starts Projects*, , pp. 80.

LVI. NUGROHO, A., VISSER, J. and KUIPERS, T., 2011. An Empirical Model of Technical Debt and Interest, *Proceedings of the 2Nd Workshop on Managing Technical Debt* 2011, ACM, pp. 1-8.

LVII. OZKAYA, I., KRUCHTEN, P., NORD, R.L. and BROWN, N., 2011. Managing Technical Debt in Software Development: Report on the 2Nd International Workshop on Managing Technical Debt, Held at ICSE 2011. *SIGSOFT Softw.Eng.Notes*, **36**(5), pp. 33-35.

LVIII. POWER, K., 2013. Understanding the impact of technical debt on the capacity and velocity of teams and organizations: Viewing team and organization capacity as a portfolio of real options, *Managing Technical Debt (MTD), 2013 4th International Workshop on* 2013, pp. 28-31.

LIX. RAMASUBBU, N. and KEMERER, C.F., 2013. Managing Technical Debt in Enterprise Software Packages.

LX. RAMASUBBU, N. and KEMERER, C.F., 2013. Towards a model for optimizing technical debt in software products, *Managing Technical Debt (MTD), 2013 4th International Workshop on* 2013, pp. 51-54.

LXI. RAMASUBBU, N. and KEMERER, C.F., 2014. Managing Technical Debt in Enterprise Software Packages. *Software Engineering, IEEE Transactions on,* **40**(8), pp. 758-772.

LXII. RUBIN, Y., KALLNER, S., GUY, N. and SHACHOR, G., 2013. Restraining technical debt when developing large-scale Ajax applications, *WEB 2013, The First International Conference on Building and Exploring Web Based Environments* 2013, pp. 13-18.

LXIII. SCHMID, K., 2013. A Formal Approach to Technical Debt Decision Making, Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures 2013, ACM, pp. 153-162.

LXIV. SCHMID, K., 2013. On the limits of the technical debt metaphor some guidance on going beyond, *Managing Technical Debt (MTD), 2013 4th International Workshop on* 2013, pp. 63-66.

LXV. SCHUMACHER, J., ZAZWORKA, N., SHULL, F., SEAMAN, C. and SHAW, M., 2010. Building Empirical Support for Automated Code Smell Detection, *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* 2010, ACM, pp. 8:1-8:10.

LXVI. SEAMAN, C. and GUO, Y., 2011. Measuring and monitoring technical debt. *Advances in Computers,* **82**, pp. 25-46.

LXVII. SEAMAN, C. and ZAZWORKA, N., 2011. Technical Debt.

LXVIII. SEAMAN, C., GUO, Y., IZURIETA, C., CAI, Y., ZAZWORKA, N., SHULL, F. and VETR\`O ANTONIO, 2012. Using Technical Debt Data in Decision Making: Potential Decision Approaches, *Proceedings of the Third International Workshop on Managing Technical Debt* 2012, IEEE Press, pp. 45-48.

LXIX. SHARMA, T., 2012. Quantifying Quality of Software Design to Measure the Impact of Refactoring, *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual* 2012, pp. 266-271.

LXX. SHULL, F., FALESSI, D., SEAMAN, C., DIEP, M. and LAYMAN, L., 2013. Technical debt: showing the way for better transfer of empirical results. *Perspectives on the Future of Software Engineering.* Springer, pp. 179-190.

LXXI. SIEBRA, C.A., TONIN, G.S., DA SILVA, F.Q.B., OLIVEIRA, R.G., ANTONIO, L.C., MIRANDA, R.C.G. and SANTOS, A.L.M., 2012. Managing technical debt in

practice: An industrial report, *Empirical Software Engineering and Measurement (ESEM), 2012 ACM-IEEE International Symposium on* 2012, pp. 247-250.

LXXII. SINGH, V., SNIPES, W. and KRAFT, N.A., 2014. A Framework for Estimating Interest on Technical Debt by Monitoring Developer Activity Related to Code Comprehension, *Managing Technical Debt (MTD), 2014 Sixth International Workshop on* 2014, IEEE, pp. 27-30.

LXXIII. STOCHEL, M.G., WAWROWSKI, M.R. and RABIEJ, M., 2012. Value-Based Technical Debt Model and Its Application, *ICSEA 2012, The Seventh International Conference on Software Engineering Advances* 2012, pp. 205-212.

LXXIV. TAKSANDE, N., 2011. Empirical study on technical debt as viewed by software practitioners. University of Maryland, Baltimore County.

LXXV. TAMBURRI, D.A., KRUCHTEN, P., LAGO, P. and VAN VLIET, H., 2013. What is social debt in software engineering? *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on*2013, pp. 93-96.

LXXVI. THEODOROPOULOS, T., HOFBERG, M. and KERN, D., 2011. Technical Debt from the Stakeholder Perspective, *Proceedings of the 2Nd Workshop on Managing Technical Debt* 2011, ACM, pp. 43-46.

LXXVII. TOM, E., AURUM, A. and VIDGEN, R., 2013. An exploration of technical debt. *Journal of Systems and Software*, **86**(6), pp. 1498-1516.

LXXVIII. TUFANO, M., PALOMBA, F., BAVOTA, G., OLIVETO, R., DI PENTA, M., DE LUCIA, A. and POSHYVANYK, D., 2015. When and Why Your Code Starts to Smell Bad.

LXXIX. VETRO', A., 2012. Using Automatic Static Analysis to Identify Technical Debt, *Proceedings of the 34th International Conference on Software Engineering* 2012, IEEE Press, pp. 1613-1615.

LXXX. YLI-HUUMO, J., MAGLYAS, A. and SMOLANDER, K., 2014. Evaluating and managing technical debt in software development lifecycle. *Product-Focused Software Process Improvement,* , pp. 26.

LXXXI. YLI-HUUMO, J., MAGLYAS, A. and SMOLANDER, K., 2014. The Sources and Approaches to Management of Technical Debt: A Case Study of Two Product Lines in a Middle-Size Finnish Software Company. **8892**, pp. 93-107.

LXXXII. YUEPU GUO, SEAMAN, C., GOMES, R., CAVALCANTI, A., TONIN, G., DA SILVA, F.Q.B., SANTOS, A.L.M. and SIEBRA, C., 2011. Tracking technical debt — An exploratory case study, *Software Maintenance (ICSM), 2011 27th IEEE International Conference on* 2011, pp. 528-531.

LXXXIII. YUEPU GUO, SEAMAN, C., ZAZWORKA, N. and SHULL, F., 2010. Domain-specific tailoring of code smells: an empirical study, *Software Engineering, 2010 ACM/IEEE 32nd International Conference on* 2010, pp. 167-170.

LXXXIV. ZAZWORKA, N., IZURIETA, C., WONG, S., CAI, Y., SEAMAN, C. and SHULL, F., 2013. Comparing four approaches for technical debt identification. *Software Quality Journal,* , pp. 1-24.

APPENDIX 2. Table of retrieved definitions

Definition Index	Source index	Definition
D1	S68	Tradeoff among quality, time, and cost
D2	S69	Technical debt is the cost difference between doing things right versus doing things fast
D3	S11	Trade off short-term gain for a longer-term cost
D4	S35	…you can't get everything you want done and therefore, you only do the things that are not necessarily the most important but the things that will give you the most payoff
D5	S11	Tradeoff between implementing some piece of software in a robust and mature way (the "right" way) and taking a shortcut which may provide short term benefits, but which has long term effects that may impede evolution and maintainability
D6	S6	Embodies the dichotomy between decisions focusing on the long-term effects to the quality of the software versus focusing on the short term effects on the time-to-market and business value of the software
D7	S6	Code smells - poorly designed areas of the software which strongly indicate a need for refactoring
D8	S12	Doing things the "quick and dirty" way
D9	S12	A design or construction approach that's expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time)
D10	S17	Problem of deciding when and how to refactor a system realization to improve its structure as a basis for future evolution

Definition Index	Source index	Definition
D11	S2	Compromises in a system in one dimension (e.g., modularity) to meet an urgent demand in some other dimension (e.g., a deadline), and that such compromises incur a "debt": on which "interest" has to be paid and which the "principal" should be repaid at some point for the long-term health of the project
D12	S4	Key trade-offs related to release and quality issues
D13	S1	Results from the tension between engineering "best practices" and other factors (ship date, cost of tools, and the skills of engineers that are available, among others)
D14	S1	Strategy to save time or money
D15	S1	All the shortcuts that save money or speed up progress today at the risk of potentially costing money or slowing down progress in the (usually unclear) future
D16	S13	It is "not quite right code" which we postpone making it right
D17	S20	Workarounds increase complexity which makes enhancements and testing more challenging
D18	S20	Technical debt is any gap within the technology infrastructure or its implementation which has a material impact on the required level of quality
D19	S14	Refers to immature work in a software system that takes compromises in one dimension to meet urgent needs in some other dimension
D20	S10	Lack of refactoring incurs technical debt to be repaid in the form of increased maintenance cost
D21	S15	Phenomenon in which technical quality issues in a software system can lead to future problems if not resolved immediately

Definition Index	Source index	Definition
D22	S15	Technical debt is defined as the cost to improve technical quality up to a level that is considered ideal. The interest of technical debt is the extra cost spent on maintaining software as a result of poor technical quality
D23	S15	Unintentional debts generally result from poor coding practices. Intentional debts, on the other hands, are debts committed with informed decisions
D24	S15	The cost of repairing quality issues in software systems to achieve an ideal quality level
D25	S18	Code smells - characterize different types of design shortcomings in code
D26	S18	Code smells - indicator that points to the violation of object-oriented design principles
D27	S7	Describes the effect of immature software artifacts on software maintenance – the potential of extra effort required in future as if paying interest for the incurred debt
D28	S19	Long term maintenance tradeoff
D29	S8	Code smells refer to commonly occurring patterns in source code that indicate poor programming practices or code decay.
D30	S8	Code smells are symptoms of real problems in source code, such as violations of coding convention, shortcuts or bad designs
D31	S8	Technical debt metaphor reflects the relationship between the short-term benefits of delaying certain maintenance tasks (or doing them quickly and less carefully) and the long-term cost of those delays
D32	S86	TD metaphor describes a tradeoff between short-term and long-term goals in software development
D33	S58	Including both intentional and unintentional violations of good architectural and coding practice

Definition Index	Source index	Definition
D34	S59	Problems or quality issues in software that will cost organizations owning the software greater expenses if the problems are not resolved
D35	S59	Technical debt is costs to repair problems in software systems in order to achieve an ideal level of quality
D36	S78	Software maintenance projects are often constrained by budget, schedule and resources. To meet these constraints, trade-offs are made throughout the software development lifecycle. While these trade-offs allow project teams to meet customer and management expectations in the short run, the compromise results in additional cost later in the maintenance process. This phenomenon is known as "technical debt"
D37	S63	Often we find it difficult to adapt to any changes during later phases of a software development project. Primary reason for this is rigidity in design and code which do not allow major changes to be incorporated. This inflexibility substantially increases the cost of post-delivery enhancement and maintenance and is termed as Technical Debt
D38	S70	Technical debt is a useful tool for identifying deviations from desired practices and for measuring improvement of the code.
D39	S77	Social debt - represents the set of strained social relationships that emerges as a consequence of debtor-creditor circumstances
D40	S8	Technical debt is the consequence of trade-offs made during software development to ensure speedy releases
D41	S8	The cost to improve software quality to an ideal level
D42	S54	Navigating a path that considers both value and cost, to focus on overall return on investment over the lifespan of the product
D43	S56	An operational liability which may incur an interest if not managed, cleared and transformed from liability to value

Definition Index	Source index	Definition
D44	S61	A situation in which developers accept quality compromises in the current release to meet a deadline (e.g. delivering a release on time)
D45	S62	Code smells - choices in object-oriented systems that do not comply with widely accepted principles of good-object oriented design
D46	S62	Modularity violation
D47	S43	Tradeoff potential longer-term benefits of thorough conformance to development and maintenance standards for the immediate short-term business benefits of rapidly adding functionality by taking shortcuts (i.e., incurring technical debt)
D48	S1	Shortcuts that save money and time today can cost you down the road.
D49	S38	Technical debt is simply defined as deferred work that is not directly related to new functionality, but necessary for the overall quality of the system
D50	S49	Product's deficiencies, caused by shortcuts or incomplete engineering knowledge, which may speed up software development and delivery, but inevitably have their drawbacks and incur additional, delayed costs
D51	S46	Aspects of the software we know are wrong, but don't have time to fix now
D52	S46	Grime: accumulation of non-pattern code in classes following a design pattern
D53	S40	The burden placed on software engineers when shortcuts taken to speed development lead to long-term production setbacks

Definition Index	Source index	Definition
D54	S24	Technical Debt is a risk to the cost and operational performance of critical business applications, and left unaddressed can so degrade an application that its benefits to the business cannot be justified by its growing costs or operational risks
D55	S24	System evolves its complexity increases unless work is done to maintain or reduce it
D56	S24	Future costs attributable to known structural flaws in production code that need to be fixed, including both principle and interest
D57	S32	Unnecessary complexity in design and code, knowingly or unknowingly
D58	S33	The rework or unfulfilled outcomes/capabilities that result from insufficiently (or poorly) engineered or implemented solutions
D59	S33	When teams overly focus on expedited engineering, taking shortcuts that impact longer term maintenance, rework, and evolvability of the system
D60	S45	Technical Debt is the gap between: Making a maintenance change perfectly (Preserving architectural design, Employing good programming practices and standards, Updating the documentation, Testing thoroughly) And making the change work As quickly as possible With as few resources as possible
D61	S43	Software development and maintenance shortcuts taken by programmers to deliver short-term business benefits
D62	S27	Technical debt is a metaphor for delayed software maintenance tasks (or doing them quickly and less carefully) on software projects

Definition Index	Source index	Definition
D63	S27	As a result, low-quality artifacts emerge, which in turn adds more constraints on future maintenance tasks and makes modification more difficult, costly and unpredictable. This phenomenon is called “Technical Debt”
D64	S27	Delayed or “quick and dirty” work in the design and implementation phases, resulting in immature code, but this metaphor has been extended to include any immature artifacts in the software development lifecycle
D65	S27	Code smells - patterns that indicate poor programming practices and bad design choices
D66	S22	Technical debt is a term that has been used to describe the increased cost of changing or maintaining a system due to shortcuts taken during its development
D67	S22	Development team takes when it opts for an easy approach to implement in the short term, but that has a great possibility of having a negative impact in the long term
D68	S22	Technical Debt includes those internal tasks you choose not to do now, but it runs a risk of causing future problems if not done
D69	S51	Code bad smells, i.e., symptoms of poor design and implementation choices
D70	S51	Trade-offs between delivering the most appropriate but still immature product, in the shortest time possible
D71	S85	Measuring architecture rework as a proxy for short-term versus long term trade-offs, a.k.a technical debt
D72	S53	Short-term goals in software development are traded for long term goals (e.g., quick-and-dirty implementation to reach a release date vs. a well-refactored implementation that supports the long term health of the project)

Definition Index	Source index	Definition
D73	S37	Various weaknesses in the design or implementation of a system resulting from trade-offs during software development usually for a quick release
D74	S48	The consequences of decisions that affect the maintenance of a software system such as decisions regarding architecture and code structure are described with attributes of financial debt
D75	S23	The problem of technical debt caused by shortcuts that were taken to increase development speed that result in code that is difficult to maintain
D76	S50	Software practitioners accept compromises in the project in some magnitude (poor documentation, poor testing, incomplete system implementation, etc.) to meet time and budget constraints. The compromises in the system incur "technical debt" which has to be repaid at some point for the long term health of the project. If the debt remains unpaid, it could incur more additional costs, in terms of software that is harder to maintain and more error prone
D77	S50	If there is anything that is still required to be done it comes as a debt that is why I consider this to be a technical debt
D78	S50	It is basically the decision whether to implement the architecture in a standard compliant way or you make a decision based on time to, I wouldn't say necessarily take shortcuts but to do the implementation as quickly as possible
D79	S50	What I would consider technical debt is, if you are implementing part of a system or updating part of a system without taking enough care to keep the system maintainable and without enough care to keep the system consistent internally
D80	S50	Without enough care to keep the system consistent internally

Definition Index	Source index	Definition
D81	S39	Issues and development risks incurred either intentionally or unintentionally throughout the entire software development process
D82	S39	Label and communicate (internal) software quality issues, its hidden risks for future development and maintenance as well as their costs incurred
D83	S39	Technical debt (TD) measures the effort necessary to clean up the code as well as the extra costs due to TD
D84	S35	Technical debt is a metaphor for the consequences that software projects face when they make trade-offs to implement a lower quality, less complete solution to satisfy business realities
D85	S35	…what shortcuts could you take that maybe gets you out to the markets faster versus, you know, that might cause you long term pain.
D86	S35	…some shortcut that has been taken or some less than desirable implementation has been done or we've de-scoped something for one reason or another and so we've acquired some amount of the work that can or should be done at some point in the future. The reason why I believe it comes into being is simple, you know, business situations. We don't have an unlimited number of resources, we don't have an unlimited amount of time, and so, the business reality forces us to make choices at points in time to be able to get the broader outcomes of, you know, delivering a solution
D87	S35	Trade-offs, which is, what is ideal versus what is practical or what is achievable, I guess is a better word, given the constraints
D88	S26	It encompasses a relationship between the decisions made and their effect on the quality of the system

Definition Index	Source index	Definition
D89	S36	Software quality gap - disagreements about the amount of time that should be used for quality improvements
D90	S41	The technical debt metaphor conceptualizes this tradeoff between short-term and long-term value: taking shortcuts to optimize the delivery of features in the short term incurs debt, analogous to financial debt, that must be paid off later to optimize long-term success
D91	S41	The term ‘technical debt’ describes an aspect of the tradeoff between short-term and long-term value in the development cycle
D92	S29	The term technical debt describes this process of accumulating hidden work
D93	S36	Model debt, which measures and communicates the software quality of the domain model
D94	S34	Metaphor was originally used to describe the problem of not improving less-than-ideal code to non-technical stakeholders; "not quite right code which we postpone making it right"
D95	S31	Technology debt - Accumulated obligation owned by current CIO (debtor) to future CIO (creditor), where previous decisions limit prospective decisions
D96	S22	Different issues during the software development life cycle, covering aspects that impact negatively on deployment activities, evolution of a system or any obstacle that hinders the progress of activities involved in its development
D97	S27	Architectural violations or deficiencies in source code; testing debt refers to the tests that are skipped or not developed sufficiently; defect debt refers to the known defects that are not fixed yet; documentation debt refers to missing or out-of-date documentation
D98	S46	Tasks that were left undone, but that run a risk of causing future problems if not completed

Definition Index	Source index	Definition
D99	S46	Design debt – any kind of anomaly or imperfection that can be identified by examining source code and/or related documentation, that leads to decreased maintainability if not remedied
D100	S44	The toll that suboptimal decisions or actions impose on the future welfare of that project
D101	S46	Incomplete, immature, or inadequate artifact in the software development lifecycle
D102	S47	The invisible results of past decisions about software that negatively affect its future
D103	S49	Technical debt typically is an internalized (engineering-based) assessment
D104	S15	The intention (deliberate or inadvertent) and awareness (reckless or prudent) of committing to a debt
D105	S38	The difference between what was delivered and what should have been delivered is known as technical debt
D106	S29	Technical debt considers the deviation between the current and optimal product state
D107	S29	Technical debt captures the uncertainty for a software project and communicates about the effects it causes